PalmSecure-iO©

Ethernet & USB

Version 4.5 Revision 2

Palm Vein Recognition Integration Platform

iCOGNIZE GmbH ©2012

System Integrator/OEM-Partner Documentation Ver. 1.3

Overview

1. Introduction 5

2. System description 5

2.1 Platform feature description 6

2.2 Platform hardware description 7

3. Feature list 11

4. Mechanical description 14

4.1 Hardware platform 14

4.2 ManuScan© Indoor solution 15

4.3 ManuScan© Outdoor solution 19

5. Operating parameters 27

5.1 Hardware platform 27

5.2 ManuScan© Indoor solution 28

5.3 ManuScan© Outdoor solution 29

6. Component and connection map 30

6.1 Installation connectors (bottom side) 30

6.2 Proprietary connectors (bottom side) 31

6.3 Jumpers (bottom side) 31

6.4 Optional LEDs (bottom side) 31

6.5 Others (bottom side) 31

6.6 Installation connectors (top side) 31

6.7 Proprietary connectors (top side) 32

6.8 Jumpers (top side) 32

6.9 Optional LEDs (top side) 32

6.10 Others (top side) 33

6.11 Installation connectors outdoor (top side) 33

6.12 Others outdoor (top side) 35

7. Electrical hardware installation 37

8. Mechanical hardware installation 39

8.1 Step-by-step mechanical installation overview 40

9. Software installation 42

10. Communication 44

10.1 Interface settings 44
10.2 Temperature sensors 44
10.3 Protocol 45
10.3.1 Description 45
10.3.2 Commands 46
A. Available GIRA Esprit© design frames 65
B. Modified GIRA Esprit© design frame 66
C. Changes to former documentation 68
C.1 Changes to document "PalmSecure-iO© Ethernet & USB Version 4.3; System Integrator/OEM-Partner Documentation Ver. 1.0" 68
C.2 Changes to document "PalmSecure-iO© Ethernet & USB Version 4.4; System Integrator/OEM-Partner Documentation Ver. 1.0" 68
C.3 Changes to document "PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.0" 68
C.4 Changes to document "PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.1" 68
C.5 Changes to document "PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.3" 68

Table of figures

Image 1: iCOGNIZE ManuScan© Indoor solution using GIRA Esprit© Edition and PalmSecure-iO©. 6
Image 2: iCOGNIZE ManuScan© indoor solution in test operation (based on PalmSecure-iO©). 8
Image 3: Schema of the iCOGNIZE PalmSecure-iO© platform (USB version)..... 9
Image 4: Schema of the iCOGNIZE PalmSecure-iO© platform (Ethernet version).....10
Image 5: iCOGNIZE ManuScan© Indoor solution in action (based on PalmSecure-iO©).....13
Image 6: Outline and dimensions of the PalmSecure-iO© break-out board.....14
Image 7: Outline and dimensions of the PalmSecure-iO© logic board. 15
Image 8: PalmSecure-iO© USB platform assembled, and separate Ethernet logic board.....15
Image 9: Cavity-wall flush-mount container dimensions. 16
Image 10: The three GIRA Esprit© series components (with 1 Euro coin for size comparison).....17
Image 11: Dimensions (all in mm) of design frame with normal and rounded corners (6mm deep). ..17
Image 12: PalmSecure-iO© hardware assembled in flush-mount container. 17
Image 13: iCOGNIZE ManuScan© Indoor solution before wall installation. 18
Image 14: ManuScan© Indoor USB solution as it would appear installed in wall (with USB cable). 18
Image 15: Schematics of the iCOGNIZE ManuScan© Outdoor device with wall mount plate. 20
Image 16: Photograph of the iCOGNIZE ManuScan© Outdoor device with wall mount plate. 21
Image 17: Dimensions (in mm) of the ManuScan© Outdoor wall mounting plate.....22
Image 18: Dimensions (in mm) of the outer bounding box for the ManuScan© Outdoor device. 23
Image 19: Dimensions (in mm) for the inner bounding box for the ManuScan© Outdoor device.....24
Image 20: Dimensions (in mm) for the cable-out bounding box for the ManuScan© Outdoor device. 25
Image 21: Use of the iCOGNIZE ManuScan© Outdoor system and the XBeam© technology in action. 26
Image 22: Pin-out and localization of the PalmSecure-iO© external connectors (bottom). 30
Image 23: Pin-out and position of several PalmSecure-iO© connectors and components (bottom)....31
Image 24: Pin-out and localization of the PalmSecure-iO© external connectors (top).....32
Image 25: Pin-out and localization of several PalmSecure-iO© connectors and components (top). 33
Image 26:Pin-out and localization of the PalmSecure-iO© outdoor external connectors (top). 35
Image 27: Pin-out and position of PalmSecure-iO© outdoor jumpers and components (top).....37
Image 28: iCOGNIZE PalmSecure-iO© demo installation scenario. 38
Image 29: Possible schematic for iCOGNIZE PalmSecure-iO© demo installation.....39
Image 30: Drill a 68mm installation hole into the cavity-wall.....41
Image 31: Prepare the hole for installation (get USB and other cables off the wall, etc.).....41
Image 32: Connect controller unit to the cables and attach it into the hole.....42
Image 33: View from backside of the wall after proper installation. 42
Image 34: GIRA Esprit© Glas Schwarz and GIRA Esprit© C Glas Schwarz (with silver bezel).....65
Image 35: GIRA Esprit© Chrome (with black and silver bezel). 65
Image 36: GIRA Esprit© Glas Mint and GIRA Esprit© C Glas Mint (with silver and black bezels).....65
Image 37: GIRA Esprit© Glas Weiss and GIRA Esprit© C Glas Weiss (with silver bezel).....65
Image 38: GIRA Esprit© Umbra and GIRA Esprit© C Umbra (with silver bezel). 66
Image 39: GIRA Esprit© Messing (with silver and black bezels). 66
Image 40: GIRA Esprit© Aluminium (with silver and black bezels). 66
Image 41: GIRA Esprit© Wengeholz (with silver and black bezels).....66
Image 42: Modified GIRA Esprit© design frame front side (aluminum here). 67
Image 43: Modified GIRA Esprit© design frame backside (aluminum here).....67

1. Introduction

With PalmSecure©, Fujitsu delivers an amazing new and high-secure solution for physical and logical access applications. PalmSecure© uses palm veins for biometric authentication, known as the most secure and safest biometric authentication variant right now. For that, Fujitsu provides partners with an USB 2.0-coupled sensor in combination with an authentication library running on a small computer based on Intel x86 technology or ARM cores (the little computer is named host device or authentication unit in this matter). This way, setting up a biometric demo system with basic functionalities is quiet easy.

Thus, to build a full-featured system which can be installed and used out-of-the-box, a lot more work has to be done. Beside the need for a mechanical solution to install the sensor at the point-of-use, there are a lot of extra-features the stand-alone sensor should be extended with to answer customer special requests and needs. For example, this is a user-feedback system, intercom and other audio functionality, maybe video support and last but not least, interfacing to already installed devices like door-controllers and RFID card readers.

With the PalmSecure-iO© platform and two ready-to-use solutions (ManuScan Indoor and ManuScan Outdoor), the iCOGNIZE GmbH can provide customers with that.

2. System description

A solution for all of them above is the PalmSecure-iO©. This small embedded platform provides a lot of interfaces and functionality very useful for the described application. The following two paragraphs will clarify what the iCOGNIZE PalmSecure-iO© device or system is about, what it can do and how it may help system integrators or OEM partners to shorten the time-to-market for own PalmSecure© products dramatically.

Basically it is a hardware platform providing the functionality needed for a full-featured device mentioned above. For maximum implementation flexibility and easiest end-user-installation potential, the hardware is designed to be pretty small and is further more modeled to fit into a standard cavity wall socket. It is even that small that there is still enough free room in the socket to embed other components like a speaker, an RFID reader module and the Fujitsu PalmSecure© sensor itself. It turns out that this opens a minimum-support installation scenario as installation process is almost identical to the installation of a light switch for example. Anyway, customer-specific hardware shapes could be made available on customer request.

Thanks to the consequent use of industrial components, the device hardware itself is fully operational over the full industrial temperature range of -35°C to 85°C (what is NOT true for the Fujitsu PalmSecure© sensor itself).

Right now, two complete and ready-to-integrate solutions using the PalmSecure-iO© platform are available from iCOGNIZE, an indoor-version (ManuScan Indoor) using GIRA Esprit© Glass frames and an outdoor version (ManuScan Outdoor) with ruggedized aluminum housing in IP65 standard with integrated heater. The outdoor version uses the patented iCOGNIZE XBeam© positioning system, a unique iCOGNIZE feature.

The image below shows the iCOGNIZE indoor-solution, the ManuScan© Indoor.

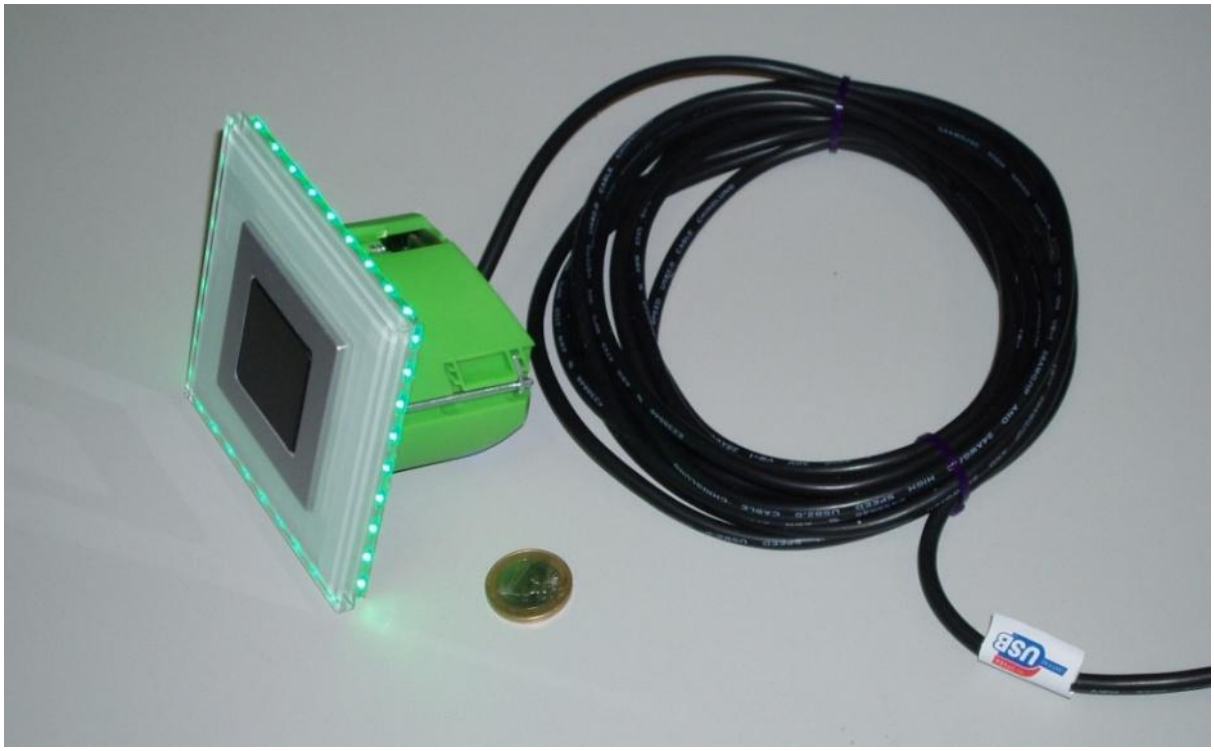

Image 1: iCOGNIZE ManuScan© Indoor solution using GIRA Esprit© Edition and PalmSecure-iO©.

2.1 Platform feature description

One of the most important features is the full-color RGB LED PWM controller handling up to 50 full-color LEDs. The LEDs are combined into 4 groups which can be switched on and off separately and thus are ideal to provide user positioning help. Over 16.7 million colors can be synthesized with this device, even light animations are possible. And all of this without using processing power from the host device. The full light sequence is running on the PalmSecure-iO©, just initiated by the host.

Additionally the PalmSecure-iO© has a beeper on board, controllable in volume, frequency and sound duration without taking host device processing power, too. That can ideally be used as extra feed-back for user interaction or applications like sabotage alert.

The sabotage alert, for example, is triggered by an integrated micro switch or an integrated G-shock sensor indicating any modification or sabotage attempt done to the system. On the other hand side, these events can be reported to the host device in an interrupt like matter. So, no host device polling is necessary.

This is also the case for the internal general purpose digital I/O bank, where four digital inputs and two open-collector outputs are available. This suits ideally for applications where a request-to-exit (REX) button or an additional convenient light is necessary. One can think of a lot of other things that could be done with this feature.

A free accessible potential free relay contact with NO and NC contacts for different purposes, several integrated temperature sensors and a 1-Wire© bus for external temperature sensors complete the system (other sensors can be implemented on customer request).

Furthermore, the platform delivers a number of well-known interfaces of building automation and security to enable a seamless integration in existing installations. Currently a version with Wiegand input, Clock/Data input, Wiegand output, Clock/Data output, RS-485/RS-232 (alternatively) interface and CAN bus interface is in production. Other interfaces could be implemented on customer request, either by just updating the firmware or by modifying the PalmSecure-iO© hardware.

Additionally there are three legacy i.e. proprietary interfaces available, which are I^2C or Two-Wire-Interface (TWI), SPI and 1-Wire©. These three interfaces are perfect to adapt extensions like internal RFID

card readers (where no RFID reader is installed already) or OLED-displays for example or and analog-to-digital converter. Even iButton® applications are thinkable.
Finally, a high-current open-drain output for heater control is available. This port is dedicated to control a current-driven heating device closely connected to the Fujitsu PalmSecure© sensor. In connection with the on-system external temperature sensor 1-Wire© interface, this is perfect to extend the PalmSecure© sensor operational temperature range to match industrial temperature range!
That way, upgrading an actual building or facility installation to palm vein recognition is pretty simple and easy and a lot of things are available and ready-to-use out-of-the-box. With all of that, the platform is a real all-you-need device.

2.2 Platform hardware description

The access to the above possibilities is solved using a power-efficient embedded ARM7 controller as central intelligence. It can be managed and operated using a simple plain-text and human readable ASCII protocol traveling over a serial port connection. The serial port itself is routed through a serial-to-USB converter and is this way available over the same interface the Fujitsu PalmSecure© sensor uses, too. It simply appears as virtual serial port on the host device.
For intercom and audio functionality an USB 2.0 audio device is integrated, too. It is supported by an on-board power amplifier for a speaker, delivering 1W of output power. An adaptive gain controlled microphone amplifier builds the input channel. The audio device itself is appearing as a standard sound card on the host device. Using this, a lot of extra-features are possible. Beside intercom, sophisticated user-feedback or additional security applications are possible. One can think of alert messages, recognized user name synthesis, audio spy and noise detection, etc.
To eliminate the need of three different USB 2.0 connections from the platform to the host device for the above functionality, a 4-port USB 2.0 low-power hub has been integrated onto the platform. That way, a single cable from the host device to the platform point-of-use is enough to access all of the features mentioned above. Here two ports are connected internally to the ARM CPU and the USB 2.0 audio device. The two others are available on an external board connector, where one of them should be reserved to connect the Fujitsu PalmSecure© sensor. The second one is available for everything the customer can think of. An USB camera module could be a good choice, where everything else is conceivable.
An interesting and again unique feature concerning the external USB 2.0 ports is the power control option. Meaning that every single of these two external USB 2.0 ports can be switched on and off independently and by commands to the ARM CPU. On the one side, this can be used to save a lot of power when connected USB device is not needed. On the other side, this is an important work-around for the Fujitsu PalmSecure© sensor, which is showing deadlocks in long-term use. Currently the only way to recover the sensor from this deadlock is to cycle it to a power off-on sequence. No other solution than that one from iCOGNIZE can provide that at the moment!
To extend the maximum USB cable length, an integrated buck-boost power converter makes sure, that a constant +5V power supply is available to all internal and external USB components. Even when long cables cause a USB power voltage drop. That is also true for the Fujitsu PalmSecure© and thus increasing the reliability, lifetime and stability of that device by far. This is one of the unique features of the iCOGNIZE PalmSecure-iO©! For sure, devices connected to the second external USB 2.0 port benefit from that feature, too. But users must not forget that there is a price for everything. In this case it means, because the complete system is powered over the USB cable, USB 2.0 maximum current consumption specifications are exceeded by far. Either the USB 2.0 port on the host must be able to deliver more current or a work-around like a Y-cable or an additional power supply at the host side must be used.
Where an USB connection or cable does not suit the customer or application needs, now there is an Ethernet option available. Here the device integrates a 10/100MBit/s USB-to-Ethernet bridge tunneling the USB packets through the network. Thus device appears as an UPnP device with static or dynamic DHCP IPv4 address in the network. On the host side, a device driver routes the incoming IP packets to a

virtual USB port. That way, no software adaption is necessary to use the Ethernet-driven device! Again an unique feature of the iCOGNIZE PalmSecure-iO© device. Needless to say, that the Ethernet-variant is powered using Power-over-Ethernet (PoE).

An iCOGNIZE ManuScan© Indoor solution assembled and in test operation and schemas of both of the available platform versions can be seen below.

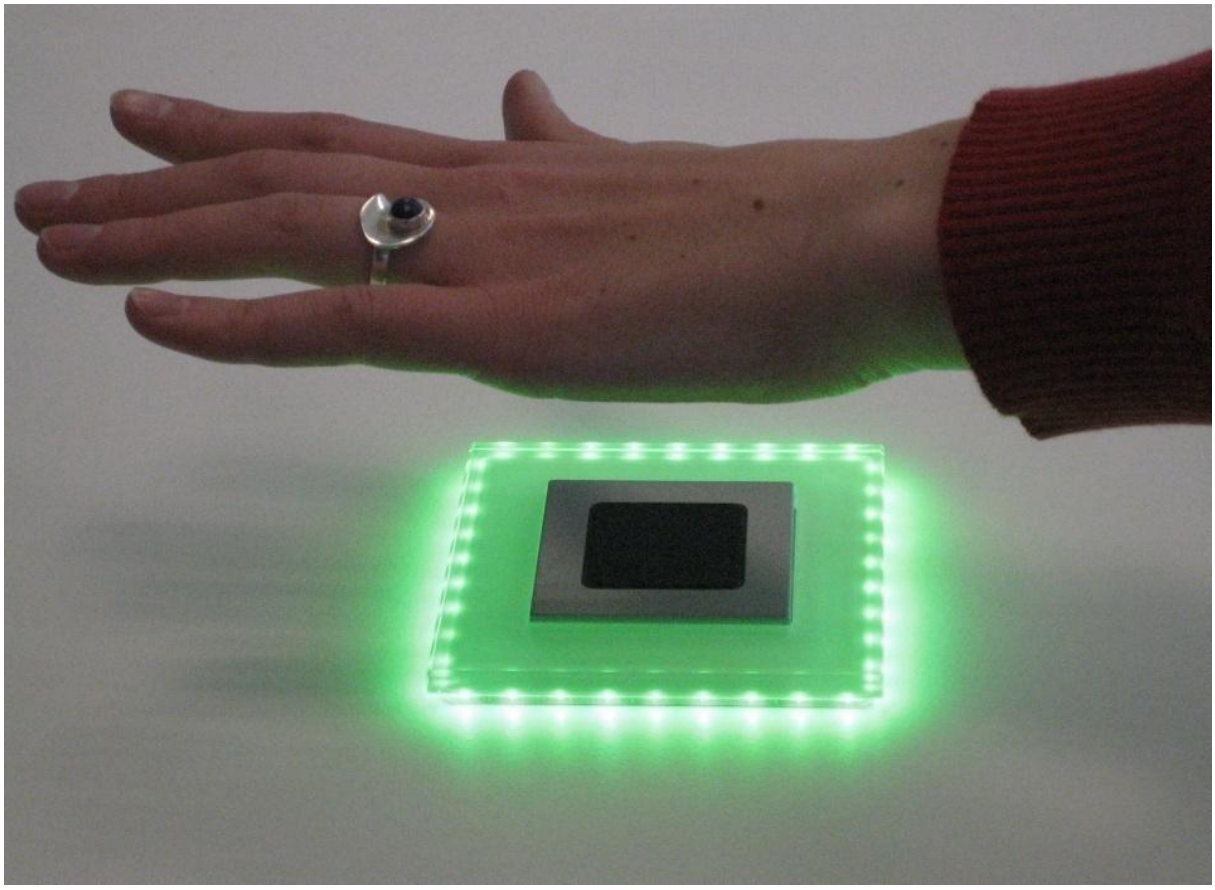

Image 2: iCOGNIZE ManuScan© indoor solution in test operation (based on PalmSecure-iO©).

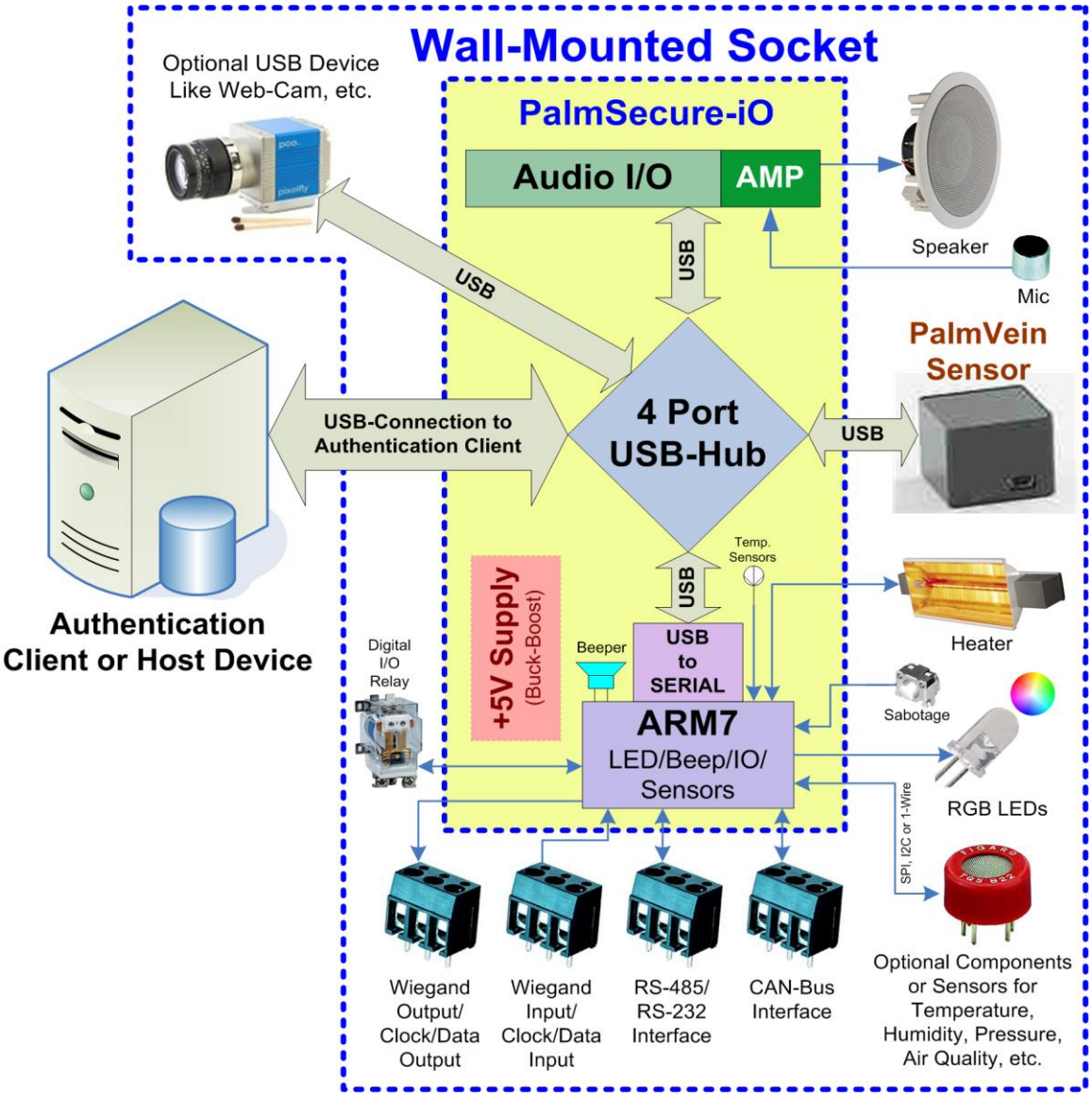

Image 3: Schema of the iCOGNIZE PalmSecure-iO© platform (USB version).

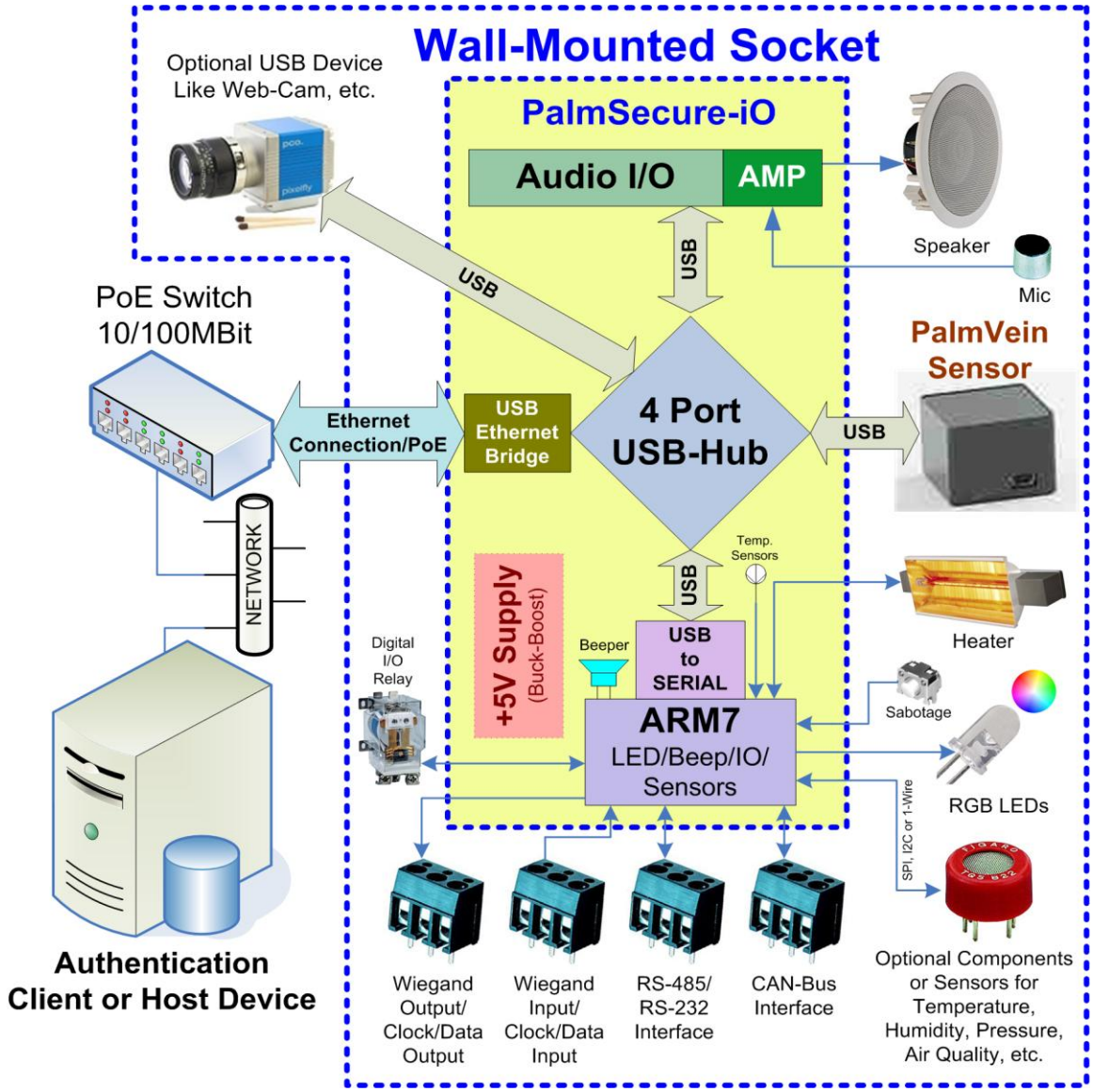

Image 4: Schema of the iCOGNIZE PalmSecure-iO© platform (Ethernet version).

3. Feature list

- Power efficient NXP ARM7 CPU running with +3.3V I/O and +1.8V core supply, 60MHz, 16kB RAM and 256kB Flash-RAM as core I/O controller.
- System control and read-out over simple plain-text human-readable ASCII serial protocol, driver provides virtual COM port on host device, asynchronous event driven messages possible for non-polling host device operation.
- CPU bootloader firmware user-upgradable over USB connection, upgrade process initiated by jumper or software commands (optional).
- CPU application firmware user-upgradable over USB connection, upgrade process initiated by software commands.
- 4-port power-efficient SMSC USB 2.0 hub integrated running with +3.3V core supply for cable reduction and USB signal conditioning.
- Supply booster in USB version make longer USB cable possible and provides stable +5V power to all components, even to externally connected USB devices. Especially for the Fujitsu PalmSecure© this increases reliability a lot (+12V supply option available).
- Ethernet-to-USB bridge with Power-over-Ethernet (PoE) as option available. PoE delivers full 15W power, according to standard IEEE802.3af.
- 3 channel full-color RGB LED PWM control for up to 16.7 million LED colors and light animations, LEDs combinable into 4 independent switchable groups for perfect user interaction and feedback (LED connection over 0.5mm pitch FFC connector).
- Volume, frequency and duration controllable beeper on board.
- Full accessible potential free DC relay contact (normally-open and normally-closed switch available) with snubber-network for spark-reduction available on board over Phoenix Combicon Compact© connector terminal.
- Sabotage switch connector (switch already integrated in GIRA Esprit© Glass frame solution) with asynchronous event signaling to host device (available over 2.54mm pitch board header connector or 0.5mm pitch FFC connector).
- G-shock sensor integrated on-board for additional sabotage attempt detection. Direction of shock detectable.
- Internal system health management using integrated temperature sensors for every critical electronic component enabling highly reliable systems.
- High-current digital open-drain output for heater control (available over 2.54mm pitch board header connector).
- 4 channel digital input for REX, door close contact, etc. Input changes can be reported as asynchronous event to host device, accessible over Phoenix Combicon Compact© connector terminal.
- 2 channel digital output (open-collector) for alert indication, etc. Outputs accessible over Phoenix Combicon Compact© connector terminal.
- I^2C, 1-Wire and SPI interfaces internally available (2.54mm board header connectors and Phoenix Combicon Compact© connector terminal) for RFID card reader module and/or different sensors (humidity, air quality, smoke, light, proximity sensor, etc.), I^2C temperature sensors already integrated, RFID card reader module integration and proximity sensor available.

- Wiegand/ClockData input interface (+5V based) available for seamless integration into existing installations, connection over Phoenix Combicon Compact© connector terminals.
- Wiegand/ClockData output interface (+5V based) available for seamless integration into existing installations, connection over Phoenix Combicon Compact© connector terminals.
- CAN bus interface (up to 1MBit/s) available for seamless integration into existing installations and connection to foreign systems. Connects over Phoenix Combicon Compact© connector terminals.
- RS-485 (RS-422) interface (up to 1MBit/s) available for seamless integration into existing installations and connection to foreign systems. Connects over Phoenix Combicon Compact© connector terminals.
- RS-232 interface (up to 1MBit/s) available for seamless integration into existing installations and connection to foreign systems. Connects over Phoenix Combicon Compact© connector terminals. This interface is only available as an alternative to the RS-485/RS-422 interface and depends on the component population of the hardware platform.
- Texas Instruments USB 2.0 audio input/output interface with on-board 1W speaker power amplifier (PA) and automatic gain controller (AGC) microphone pre-amplifier (for electrets condenser microphones with or without phantom supply) for intercom solutions and user-feedback applications, driver provides standard sound card on host system. Microphone is connected using 0.5mm pitch FFC connector, speaker is either already integrated or connection is available using 2.54mm pitch header connector.
- Fujitsu PalmSecure© sensor connection over software power-controllable USB 2.0 port for maximum flexibility. External USB 2.0 ports are available using 1.27mm semi-pitch Tyco Micromatch© connectors.
- Spare USB 2.0 port is available for system extensions like USB-camera, etc. with easy power control using software commands. External USB 2.0 ports are available using 1.27mm semi-pitch Tyco Micromatch© connectors.
- Power supply booster for USB version provides real +12V voltage for external components, even when the system is powered by +5V supply. +12V are available on the device in Ethernet version, too. Thus easy supplying of already installed components is possible. Power is available over Phoenix Combicon Compact© connector terminals.
- System communication over single USB 2.0 connection.
- System powered over single USB 2.0 connection (but with exceeding USB specifications).
- Optional SMD LEDs for error diagnosis and debugging.
- Small mechanical dimensions, fits into flush-mounted container thus making installation pretty simple and quick.

- Complete indoor and outdoor solutions using iCOGNIZE PalmSecure-iO© are available. Where the indoor version (ManuScan© Indoor) is build on top of the GIRA Esprit© light switch edition providing a maximum in elegance. The outdoor version (ManuScan© Outdoor) comes with a robust ruggedized metal housing, water-resistant (IP65) and proofed to work in the industrial temperature range. An integrated sensor heater and the patented iCOGNIZE XBeam© technology for contactless positioning complete the system.
- Platform electronics fully comply with the industrial temperature range of -35°C up to +85°C.
- RoHS conform.
- CE certified.

Image 5: iCOGNIZE ManuScan© Indoor solution in action (based on PalmSecure-iO©).

4. Mechanical description

4.1 Hardware platform

Basically and in the default configuration, the iCOGNIZE PalmSecure-iO© consists of two 4-layer PCBs. The two different PCBs are 1.55mm thick each, plated-through with green solder stop and no buried or micro vias. Both are RoHS conform. They are stacked together in a piggy-pack style with a distance of 12mm and use two mezzanine board-to-board connectors with 0.635mm pitch for that. One comes with 20 pins, one comes with 40 pins. Each is assembled on the opposite side of the PCB. Two screws, again each placed on the opposite side of the PCB, tighten the two PCBs together and by that make the connection resistive against mechanical stress. The mechanical outlines of the two PCBs are designed the way that they perfectly fit into the already mentioned cavity-wall flush-mount socket.

One of these PCBs carries the connectors for connecting external components. It is called the break-out board. Here the connectors are placed the way that they perfectly match into the break-out holes of the used cavity-wall flush-mount socket. Furthermore it holds all components for power control and power supplies. This means, for example, the PoE power supply in the Ethernet version or the +5V buck-boost DC/DC converter and the +12V boost DC/DC converter in the USB version.

One can see the dimensional drawings of the break-out board below in the scale 1:1.

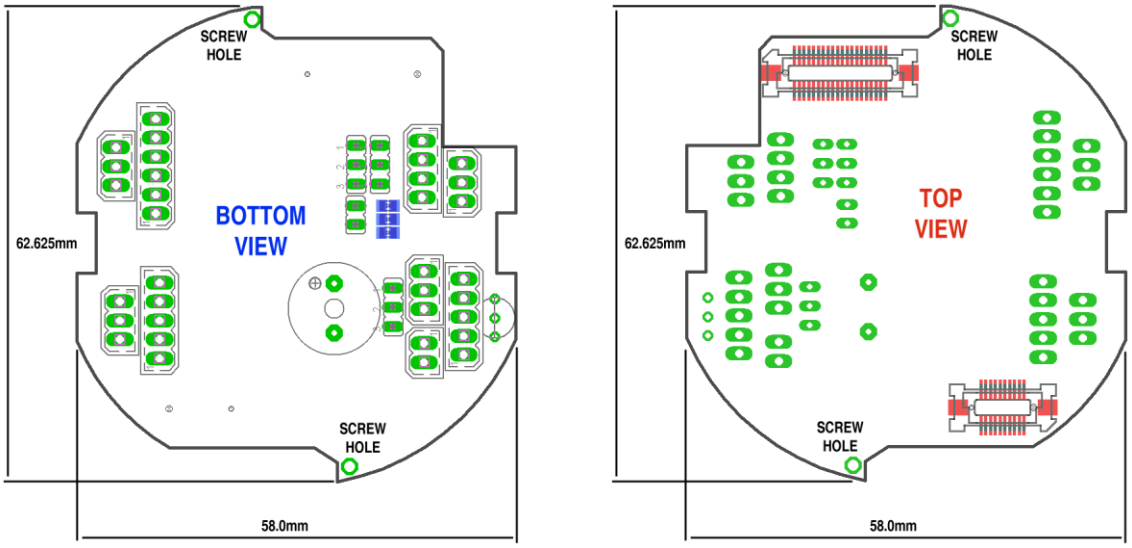

Image 6: Outline and dimensions of the PalmSecure-iO© break-out board.

The second PCB carries all the logic needed to provide the huge amount of features mentioned above and in the former paragraphs. Thus it is called the logic board. Here the USB hub, the USB-to-Serial converter, the USB-to-Ethernet bridge, LED PWM and power switches and finally the ARM CPU itself finds it place. Furthermore the FFC connector to connect the full-color RGB LEDs and some legacy or proprietary connectors and jumpers are available here (for microphone, heater connection, etc.). The board can carry the speaker for the USB-Audio I/O device, too. For sure it's outlines exactly match the break-out board and thus also fitting into the cavity-wall flush-mount container.

One can see the dimensional drawings of the logic board below in the scale 1:1.

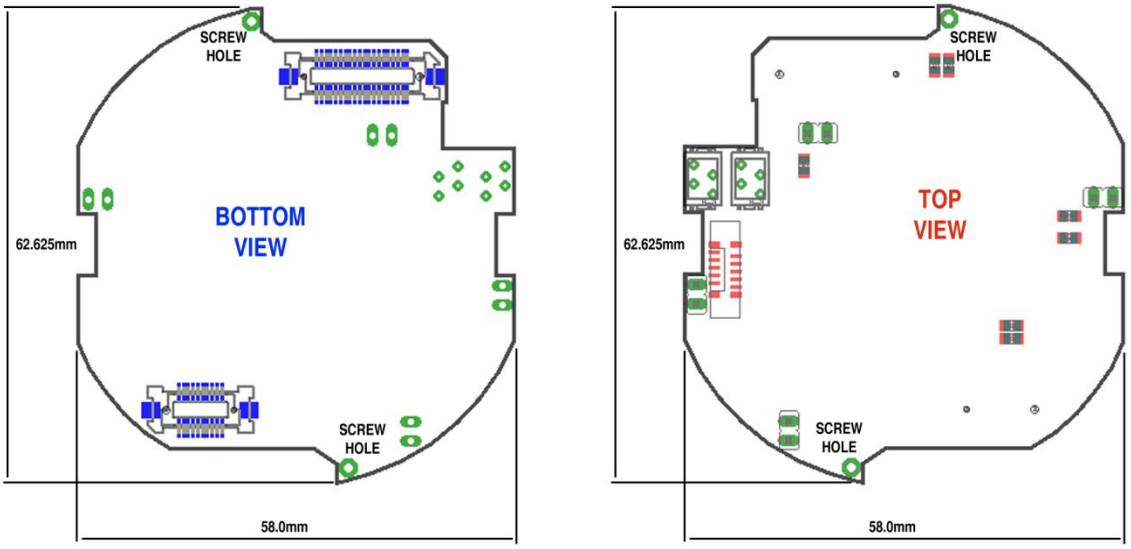

Image 7: Outline and dimensions of the PalmSecure-iO© logic board.

Below the PalmSecure-iO© platform solution can be seen in real. On the left hand side a complete and assembled USB version of the iCOGNIZE PalmSecure-iO© hardware is shown. The solution consists of the logic and the break-out board (with all its black Phoenix Combicon Compact© connectors). The latter is on the top, the former is almost completely covered by the break-out board. The tightening golden screws and the black, round beeper can be seen, too. For size comparison a 1 Euro coin has been added to the picture. On the right hand side, the break-out board of the Ethernet version is represented, carrying the same connectors as the USB version does.

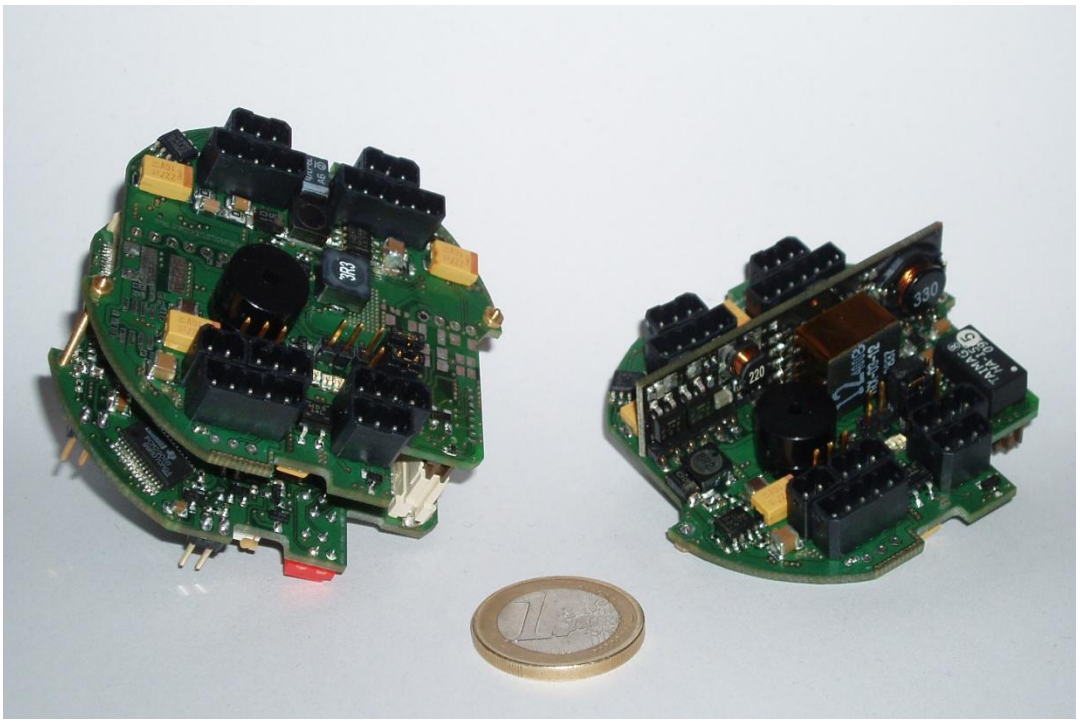

Image 8: PalmSecure-iO© USB platform assembled, and separate Ethernet logic board.

4.2 ManuScan© Indoor solution

To provide customers with a ready-to-use system for indoor installation, iCOGNIZE has developed an elegant and well-designed solution based on a standard cavity-wall flush-mount container and the GIRA

Esprit© switch installation series, called ManuScan© Indoor. With that iCOGNIZE partners are equipped with a fully-functional hardware system platform to enter the market with a palm vein re-cognition product in a very reduced amount of time. On the other hand side mechanical installation will become as easy as installing a light switch. Any this is done any other day in facility management.
Below the mentioned cavity-wall flush-mount container and the appropriate dimensions are shown.

Image 9: Cavity-wall flush-mount container dimensions.

The GIRA Esprit© switch installation series basically consists of three components. First component is an installation frame used to couple the flush-mount container to the other two components. The second part is a design cover frame made from different material (metal, glass, wood, etc.) and shapes. See appendix A for the available designs and materials. The last part is a snap-in inlay carrying the Fujitsu PalmSecure© sensor. This part tightens the three components together by just clicking into the installation frame when pushing it into it. Thus, once the flush-mount socket is installed, no more screw driver or other tools are needed to finish installation. Pressing the sensor holder into the installation frame is enough. Additionally a quick and easy change of the interior design without complicated installation processes is possible by just changing the design frame.
Because disassembling is just as well simple, a sabotage switch has been included into the design frame to indicate unauthorized modifications to the system. Furthermore, the design frame carries the full-color RGB LEDs and a microphone for audio intercom operation or other applications the user can think of.
Below the three different components can be seen disassembled. The next image shows the dimensions of the GIRA Esprit© design frame. The next three pictures show the system assembled.

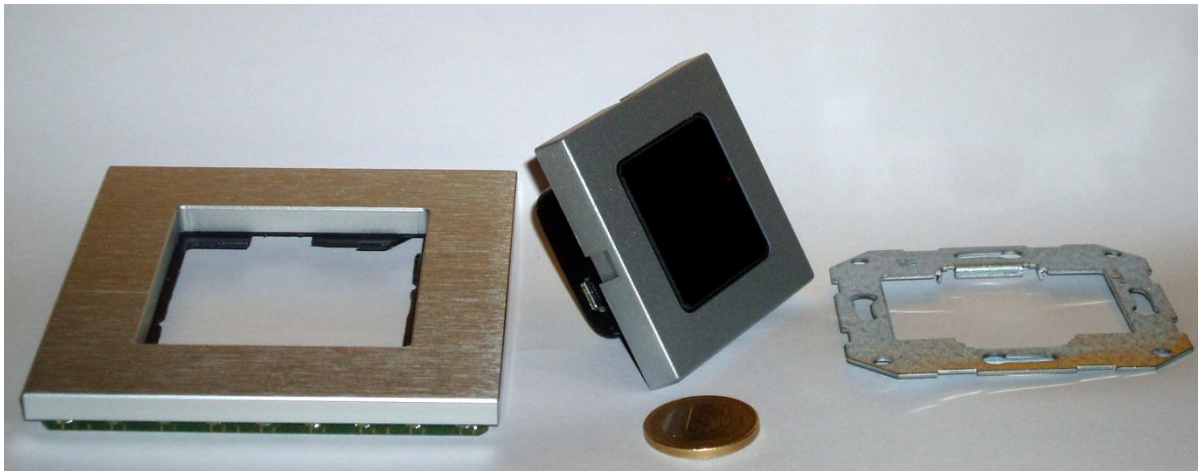

Image 10: The three GIRA Esprit© series components (with 1 Euro coin for size comparison).

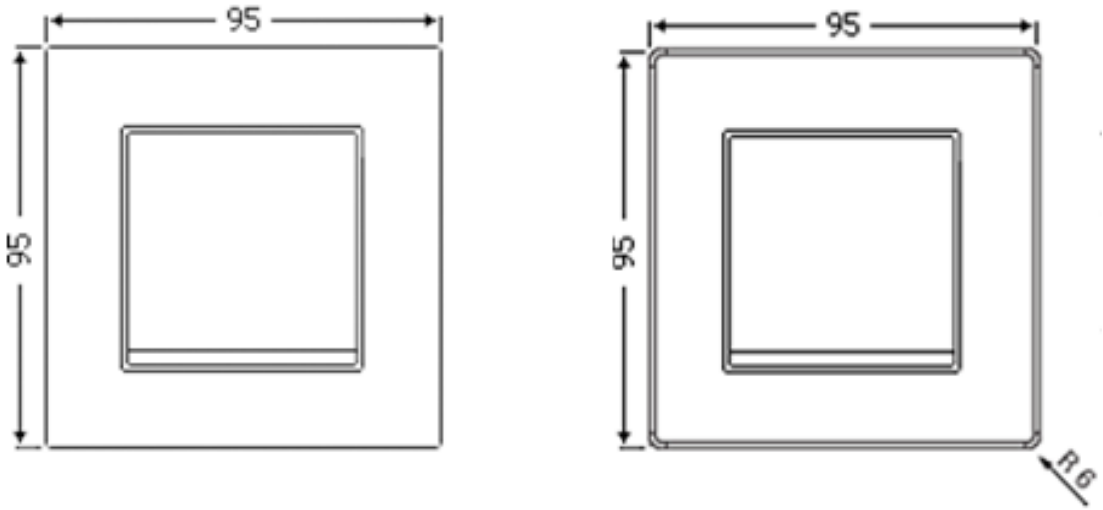

Image 11: Dimensions (all in mm) of design frame with normal and rounded corners (6mm deep).

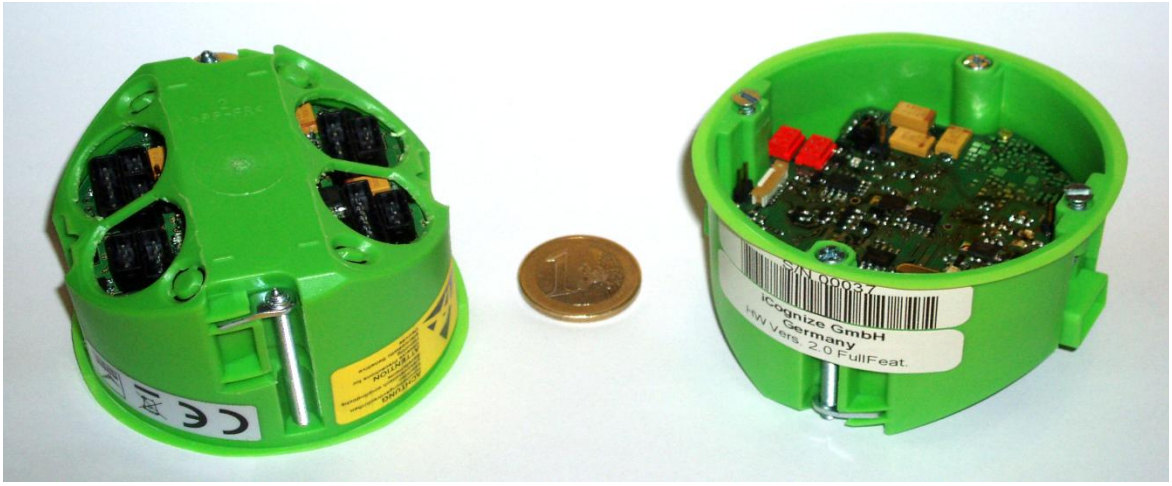

Image 12: PalmSecure-iO© hardware assembled in flush-mount container.

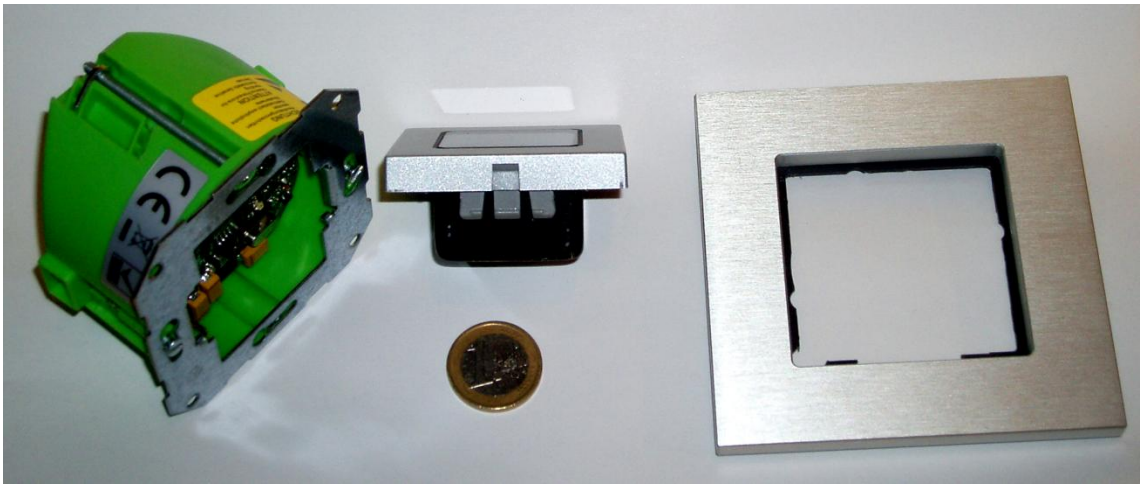

Image 13: iCOGNIZE ManuScan© Indoor solution before wall installation.

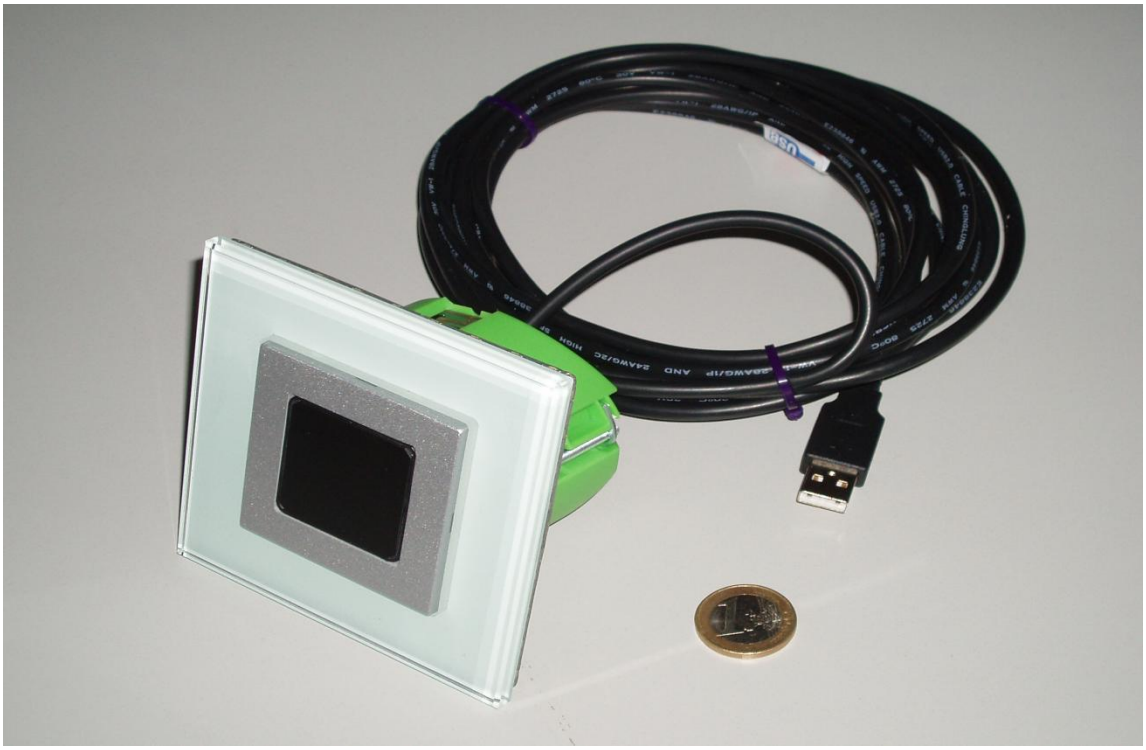

Image 14: ManuScan© Indoor USB solution as it would appear installed in wall (with USB cable).

4.3 ManuScan© Outdoor solution

Market research showed that there is a non-negligible need for a well-working outdoor version of a palm vein recognition system. And until today, no other solution is fulfilling that. iCOGNIZE decided to not leave customers alone with that requirement and developed a ready-to-use solution for that, too. Numerous tests and comprehensive experience with the technology of vein recognition made clear pretty fast that this will be a hard challenge.

First thing to deal with is the extended or industrial temperature range. Because the operating temperature range of the used Fujitsu PalmSecure© sensor is only commercial, special precautions have to be made to extend the sensors working temperature range. Basically a heater has to be integrated into the device.

Furthermore, as the system is an active optical system, sunlight is an enormous problem. To come around that problem, in the iCOGNIZE ManuScan© Outdoor system the sensor is looking down to the ground from where minimum ambient and direct light is going to be received. That DownView© called technique is patented by the iCOGNIZE GmbH. Additionally mechanical design reduces direct light exposure supplemental.

To keep the contact-less operation of the Fujitsu PalmSecure© sensor and giving the user a perfect positioning help all the same, iCOGNIZE has developed a unique and patented technique called XBeam©. That differentiates the iCOGNIZE system from market competitors who just add mechanical bearing or other constructive things to help the user with hand positioning. But here the contactless operation is no longer available. The iCOGNIZE XBeam© technology works with light pattern projecttion instead. This provides perfect positioning help by simultaneously keeping the contact-less operation mode, a unique feature of the iCOGNIZE ManuScan© Outdoor solution.

Finally the housing is made from a ruggedized aluminum block and constructed to fulfill water-resistant level of IP65. It can be easily installed to any wall or to any mounting body like post boxes, intercom installations, etc. Thus the system is well prepared for outdoor use.

The following two pictures show the iCOGNIZE ManuScan© Outdoor device in schematic manner and in real. The next four images show all the important dimensions of the iCOGNIZE ManuScan© Outdoor device. All dimensions are given in Millimeters. The last image shows the iCOGNIZE ManuScan© Out-door solution in use and the XBeam© technology in action. One can see the projection of the XBeam© system to the users hand to help positioning the user's hand.

Image 15: Schematics of the iCOGNIZE ManuScan© Outdoor device with wall mount plate.

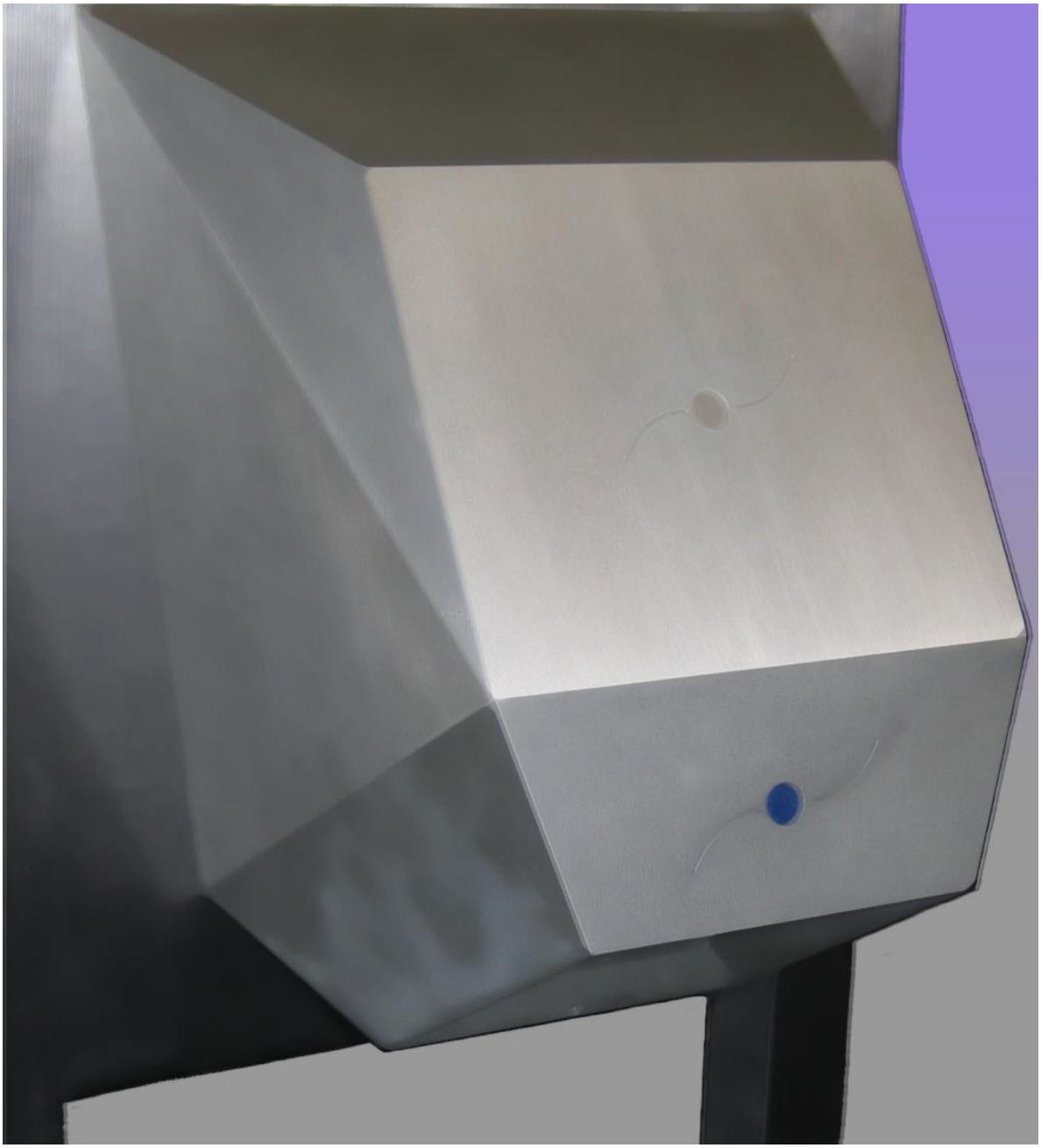

Image 16: Photograph of the iCOGNIZE ManuScan© Outdoor device with wall mount plate.

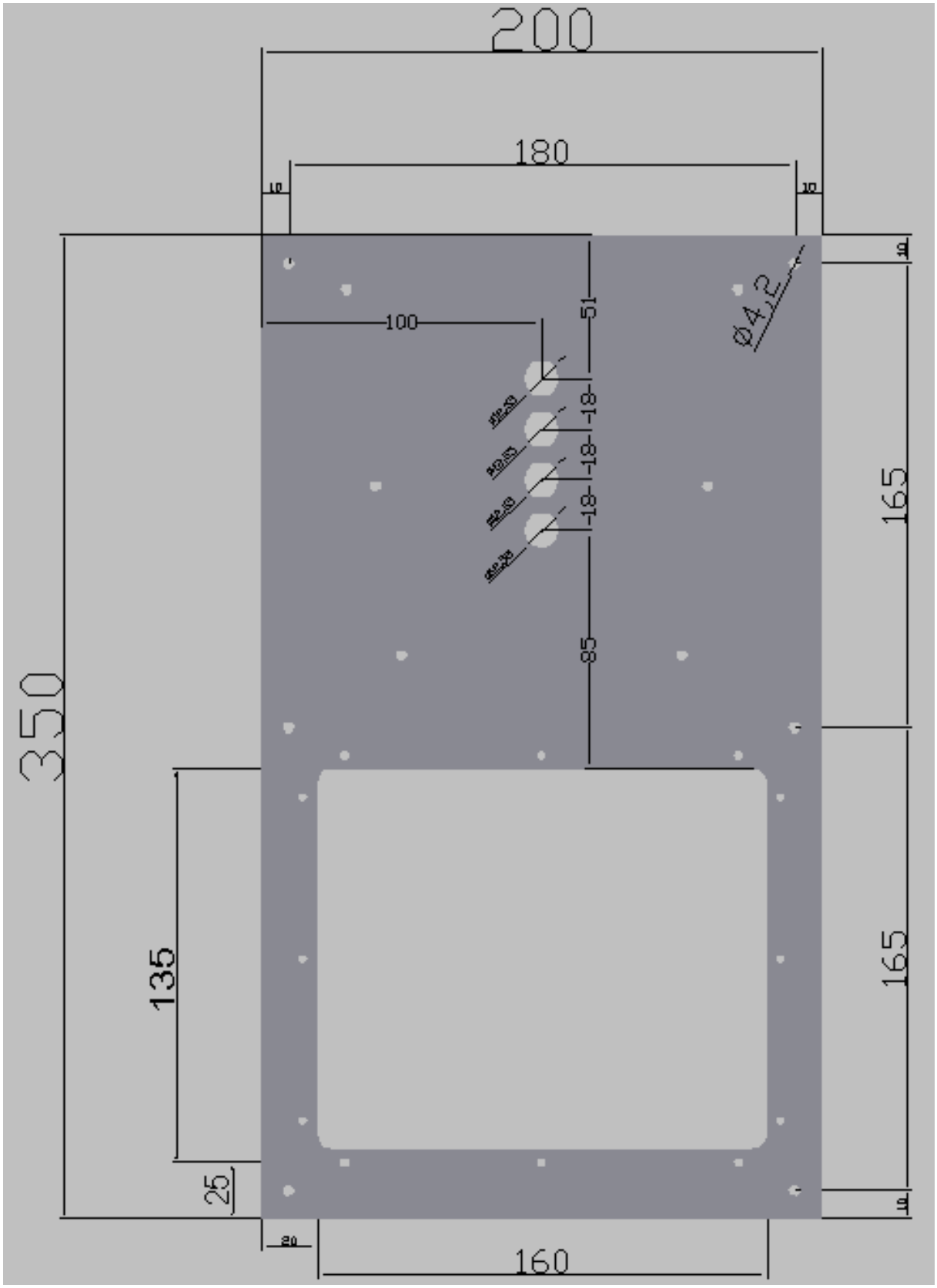

Image 17: Dimensions (in mm) of the ManuScan© Outdoor wall mounting plate.

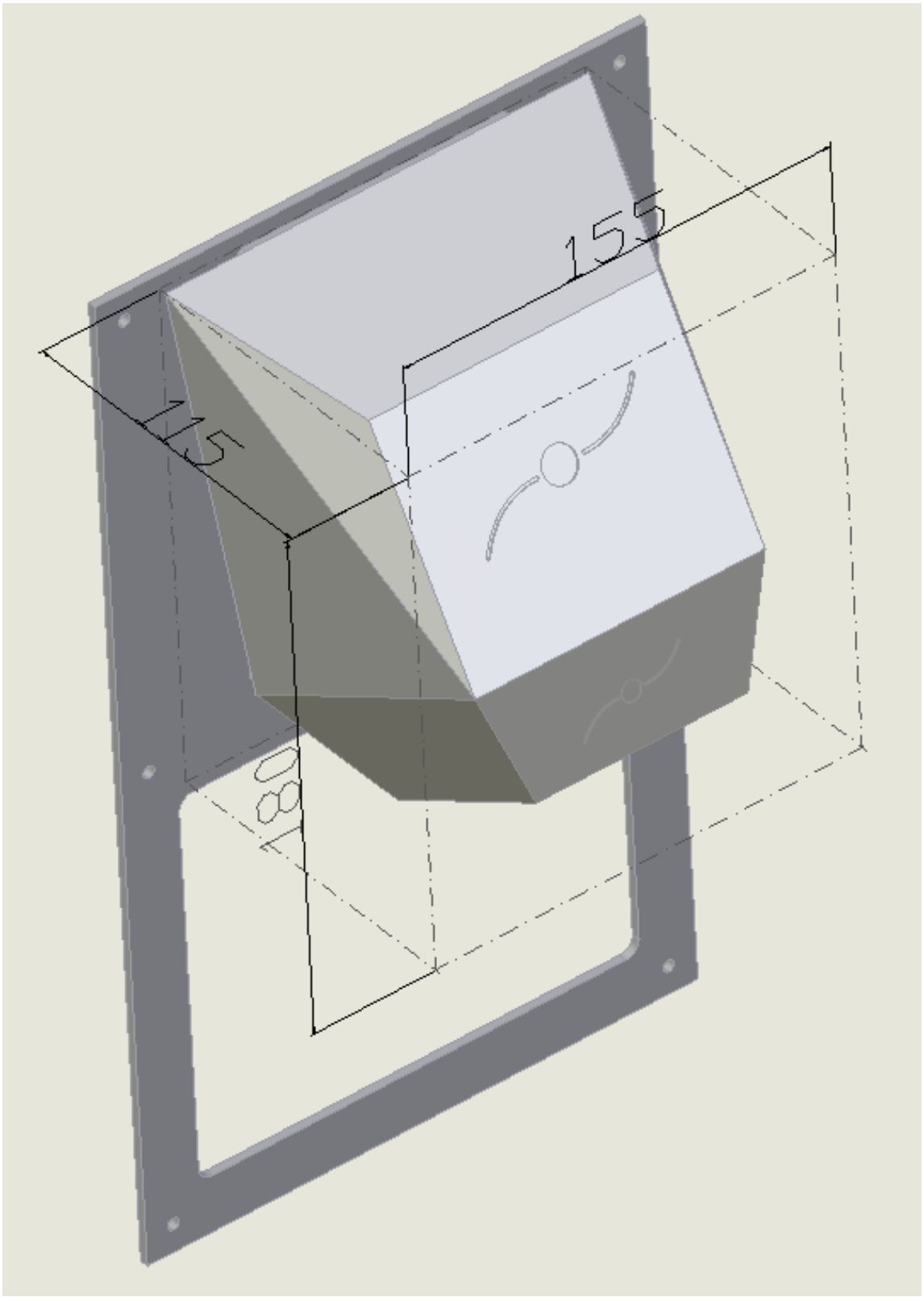

Image 18: Dimensions (in mm) of the outer bounding box for the ManuScan© Outdoor device.

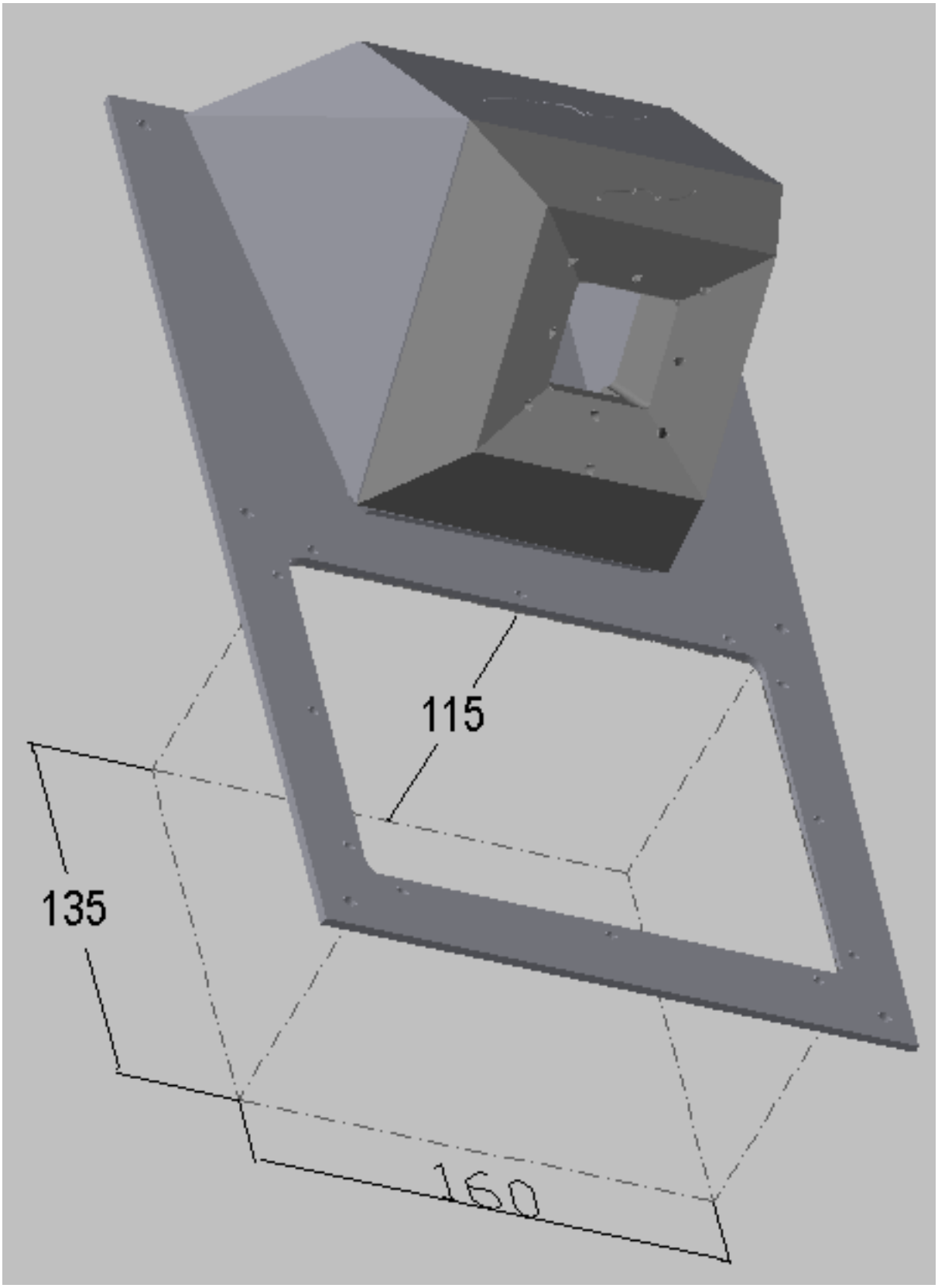

Image 19: Dimensions (in mm) for the inner bounding box for the ManuScan© Outdoor device.

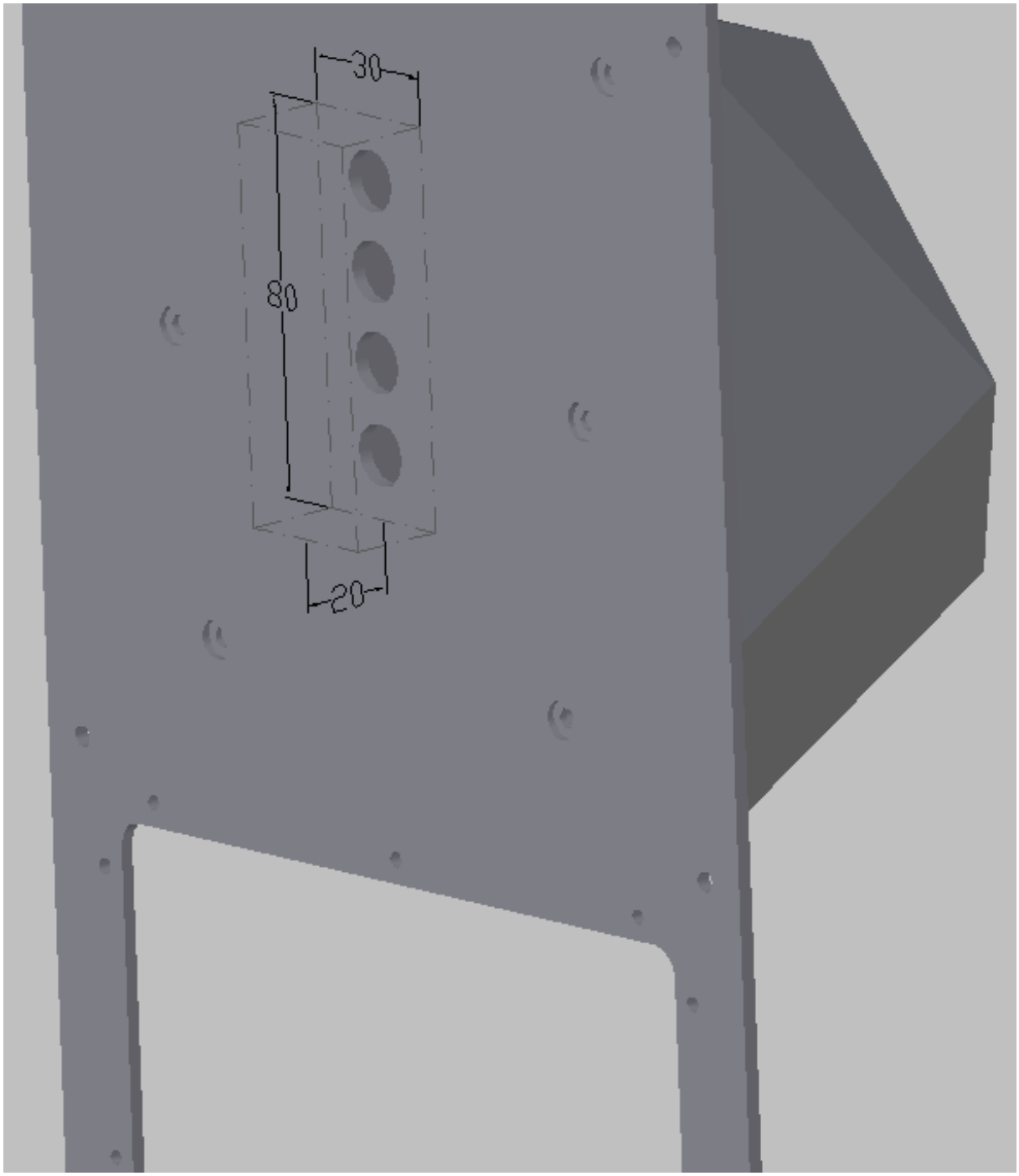

Image 20: Dimensions (in mm) for the cable-out bounding box for the ManuScan© Outdoor device.

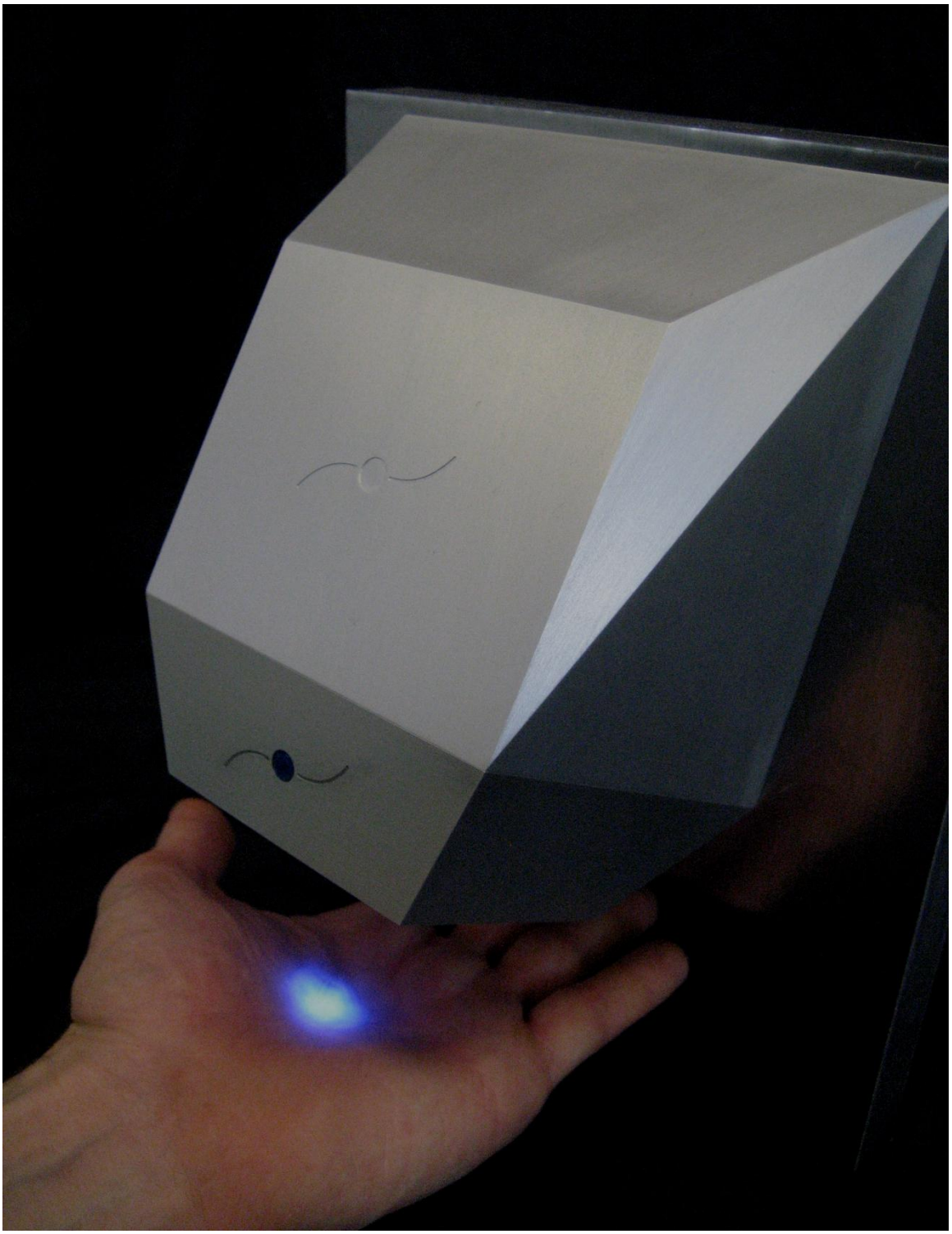

Image 21: Use of the iCOGNIZE ManuScan© Outdoor system and the XBeam© technology in action.

5. Operating parameters

5.1 Hardware platform

Storage temperature:	*-45°C to +120°C* *(-20°C to +70°C for Fujitsu PalmSecure© sensor).*
Operation temperature:	*-35°C to +80°C* *(0°C to +60°C for Fujitsu PalmSecure© sensor without heater).*
Relative humidity:	*0% to 100% (non-condensing).*
Ambient Light:	*Electronics insensitive to normal light, no direct sun light exposure;* *no direct sunlight to Fujitsu PalmSecure© sensor;* *max. 10.000 Lux diffuse light for Fujitsu PalmSecure© sensor* *(equates approximately brightest sunlight in shadow).*
Power supply (option 1):	*+5.0VDC typical (min. +3.8VDC to max. +5.5VDC);* *provided over USB or external power supply;* *polarity insensitive.*
Power supply (option 2):	*+12.0VDC typical (min. +7.5VDC to max. +13.5VDC);* *provided over external power supply;* *polarity insensitive.*
Power supply (option 3):	*+48.0VDC typical (min. +30.0VDC to max. +56.0VDC);* *provided over Power-over-Ethernet (PoE) IEEE 802.3af;* *polarity insensitive.*
Power dissipation:	*typical 0.5W for platform electronics (1.0W peak),* *typical 0.5W for speaker (1.0W peak),* *typical 2.0W for Fujitsu PalmSecure© sensor (2.5W peak).*
Additional power dissipation:	*depending on optional components and* *number or full-color RGB LEDs.*
USB up-link:	*full-speed (12MBit/s) or high-speed (480MBit/s).*
External USB down-link 1:	*full-speed (12MBit/s) or high-speed (480MBit/s);* *separate enable functionality.*
External USB down-link 2:	*full-speed (12MBit/s) or high-speed (480MBit/s);* *separate enable functionality.*
Mechanical shock:	*+/- 10.0g for electronics, unknown yet for Fujitsu PalmSecure© sensor.*
Electrical shock:	*precaution for ESD discharges of up to +/- 1.0kVDC* *on most external connectors.*
Supported OS:	*Windows© XP Pro, Windows© 2000 Pro, Windows© 7, Linux.*

5.2 ManuScan© Indoor solution

Storage temperature:	*-20°C to +70°C.*
Operation temperature:	*0°C to +60°C (with minimum air-flow in flush mount container).*
Relative humidity:	*0% to 100% (non-condensing).*
Ambient Light:	*Electronics insensitive to normal light, no direct sun light exposure; no direct sunlight to Fujitsu PalmSecure© sensor; max. 10.000 Lux diffuse light for Fujitsu PalmSecure© sensor (equates approximately brightest sunlight in shadow).*
Power supply (option 1):	*+5.0VDC typical (min. +3.8VDC to max. +5.5VDC); provided over USB or external power supply; polarity insensitive.*
Power supply (option 2):	*+12.0VDC typical (min. +7.5VDC to max. +13.5VDC); provided over external power supply; polarity insensitive.*
Power supply (option 3):	*+48.0VDC typical (min. +30.0VDC to max. +56.0VDC); provided over Power-over-Ethernet (PoE) IEEE 802.3af; polarity insensitive.*
Power dissipation:	*typical 0.5W for platform electronics (1.0W peak), typical 0.5W for speaker (1.0W peak), typical 2.0W for Fujitsu PalmSecure© sensor (2.5W peak), typical 5.0W for full-color RGB LEDs (9.0W peak).*
Additional power dissipation:	*depending on optional components.*
USB up-link:	*full-speed (12MBit/s) or high-speed (480MBit/s).*
External USB down-link 1:	*full-speed (12MBit/s) or high-speed (480MBit/s); separate enable functionality.*
External USB down-link 2:	*full-speed (12MBit/s) or high-speed (480MBit/s); separate enable functionality.*
Mechanical shock:	*+/- 10.0g for electronics, unknown yet for Fujitsu PalmSecure© sensor.*
Electrical shock:	*precaution for ESD discharges of up to +/- 1.0kVDC on most external connectors.*
Supported OS:	*Windows© XP Pro, Windows© 2000 Pro, Windows© 7, Linux.*

5.3 ManuScan© Outdoor solution

Storage temperature:	*-20°C to +70°C.*
Operation temperature:	*-35°C to +85°C.*
Relative humidity:	*0% to 100% (non-condensing).*
Water resistance:	*IP65.*
Ambient Light:	*Electronics insensitive to normal light, no direct sun light exposure; no direct sunlight to Fujitsu PalmSecure© sensor; max. 50.000 Lux diffuse light for Fujitsu PalmSecure© sensor.*
Power supply (option 1):	*+5.0VDC typical (min. +3.8VDC to max. +5.5VDC); provided over USB or external power supply; polarity insensitive.*
Power supply (option 2):	*+12.0VDC typical (min. +7.5VDC to max. +13.5VDC); provided over external power supply; polarity insensitive.*
Power supply (option 3):	*+48.0VDC typical (min. +30.0VDC to max. +56.0VDC); provided over Power-over-Ethernet (PoE) IEEE 802.3af; polarity insensitive.*
Power dissipation:	*typical 0.5W for platform electronics (1.0W peak), typical 0.5W for speaker (1.0W peak), typical 2.0W for Fujitsu PalmSecure© sensor (2.5W peak), typical 1.5W for full-color RGB LEDs (2.5W peak), typical 7.5W for heater (10W peak).*
Additional power dissipation:	*depending on optional components.*
USB up-link:	*full-speed (12MBit/s) or high-speed (480MBit/s).*
External USB down-link 1:	*full-speed (12MBit/s) or high-speed (480MBit/s); separate enable functionality.*
External USB down-link 2:	*full-speed (12MBit/s) or high-speed (480MBit/s); separate enable functionality.*
Mechanical shock:	*+/- 10.0g for electronics, unknown yet for Fujitsu PalmSecure© sensor.*
Electrical shock:	*precaution for ESD discharges of up to +/- 1.0kVDC on most external connectors.*
Supported OS:	*Windows© XP Pro, Windows© 2000 Pro, Windows© 7, Linux.*

6. Component and connection map

6.1 Installation connectors (bottom side)

1 x USB 2.0 uplink/upstream or 10/100BaseT *(CON1, 4-pin, Phoenix Combicon Compact© header).*
1 x Relay normally-open/normally-closed *(CON2, 5-pin, Combicon Compact© header).*
2 x +50VDC tolerant/0.5A open-collector digital output *(CON2, 5-pin, Combicon Compact© header).*
4 x +24VDC tolerant Digital input *(CON7, 6-pin, Combicon Compact© header).*
1 x +5VDC tolerant Wiegand input *(CON3, 3-pin, Combicon Compact© header).*
1 x +5VDC tolerant Wiegand output *(CON4, 3-pin, Combicon Compact© header).*
1 x +5VDC CAN bus interface *(CON6, 3-pin, Combicon Compact© header).*
1 x RS-422/RS-485 or optional RS-232 interface *(CON5, 3-pin, Combicon Compact© header).*
1 x +5VDC tolerant SPI interface/Interrupt *(CON8, 5-pin, Combicon Compact© header).*
1 x +12VDC/0.5A Power supply *(CON9, 2-pin, Combicon Compact© header).*

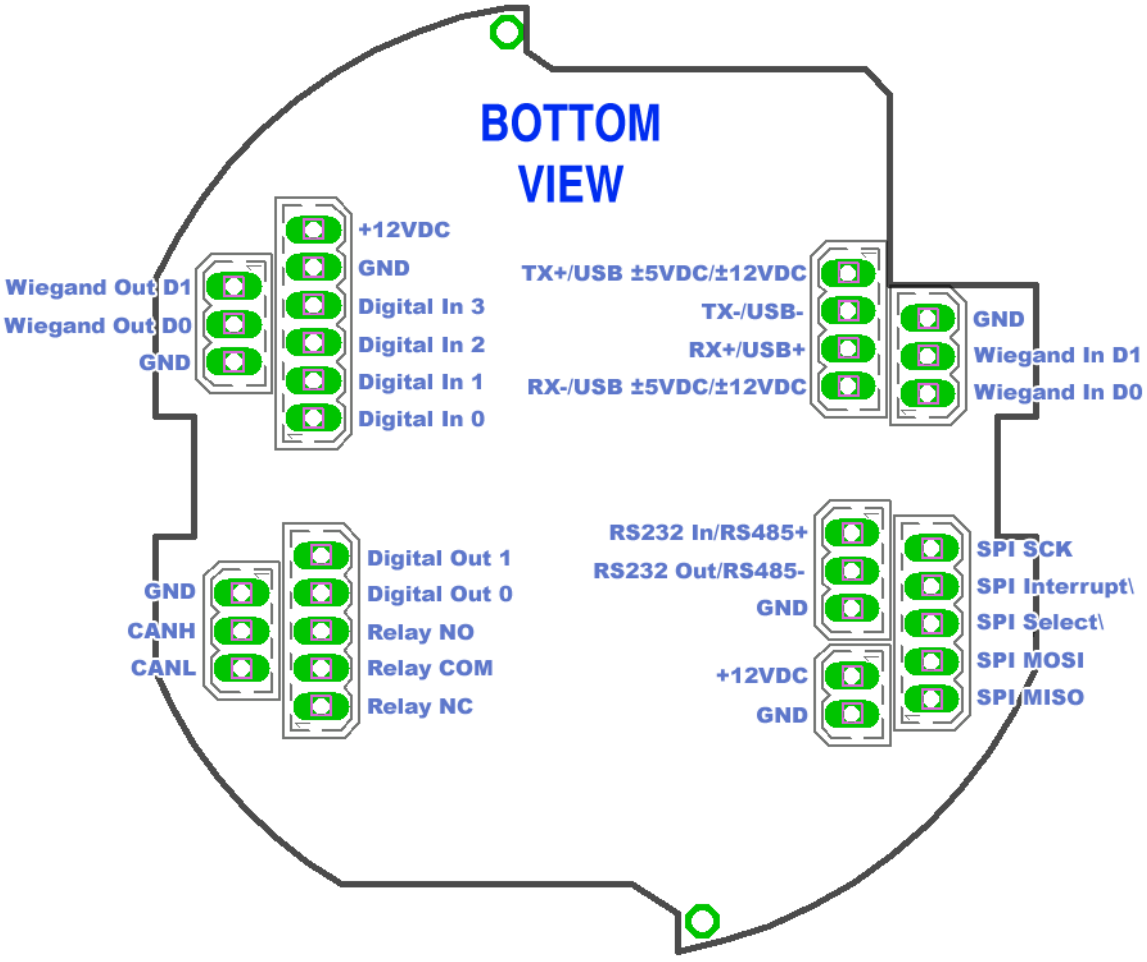

Image 22: Pin-out and localization of the PalmSecure-iO© external connectors (bottom).

6.2 Proprietary connectors (bottom side)

1 x +5VDC tolerant I²C/TWI interface/Interrupt *(JP3, 3-pin, 2.54mm pitch, single row, pin connector).*

1 x +5VDC/0.3A Power supply *(JP4, 2-pin, 2.54mm pitch, single row, pin connector).*

1 x +5VDC tolerant 1-Wire interface *(IC5, 3-pin TO92 style solder vias).*

6.3 Jumpers (bottom side)

2 x Power option selection *(JP1 and JP2, 3-pin, 2.54mm pitch, single row, pin connector).*

6.4 Optional LEDs (bottom side)

1 x +3.3VDC indication LED *(D7, bright green).*

1 x +5.0VDC indication LED *(D12, bright green).*

1 x +12.0VDC indication LED *(D13, bright green).*

6.5 Others (bottom side)

1 x PCB-mounted sound transducer *(SP1).*

1 x PCB-mounted I²C/TWI temperature sensor *(IC1).*

1 x PCB-mounted 1-Wire silicon serial number *(IC4).*

1 x PCB-mounted polyfuse for +12VDC supply option *(F1).*

1 x PCB-mounted polyfuse for PoE and +5VDC USB supply option *(F2).*

1 x PCB-mounted polyfuse for +5VDC USB supply to +12VDC booster option *(F3).*

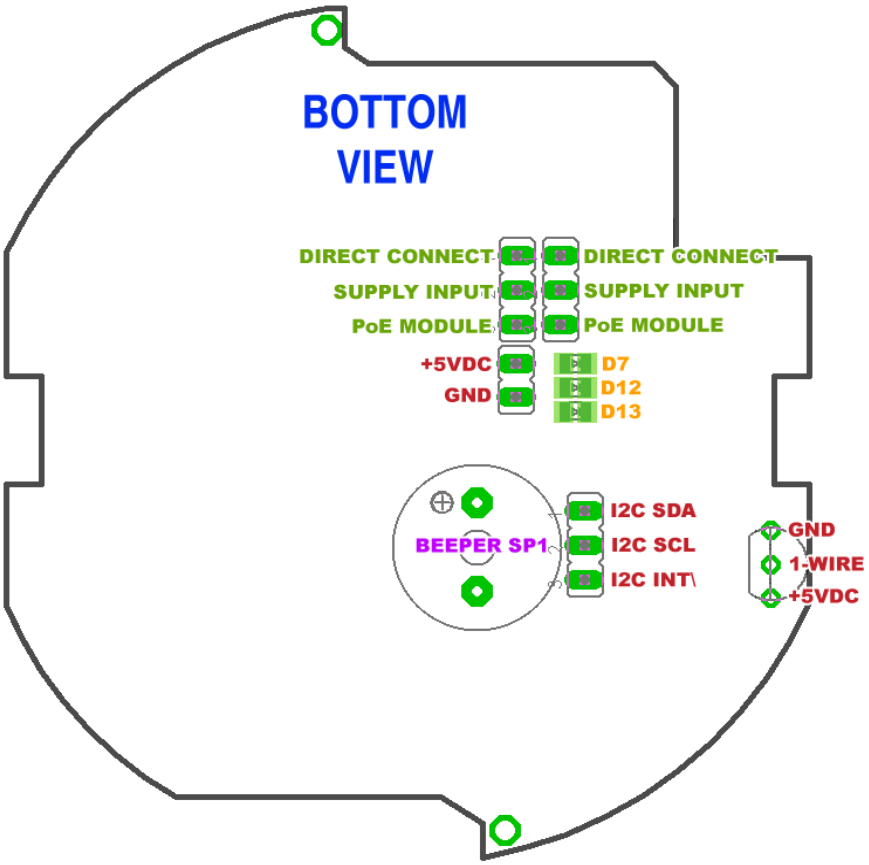

Image 23: Pin-out and position of several PalmSecure-iO© connectors and components (bottom).

6.6 Installation connectors (top side)

1 x USB 2.0 downlink/downstream (X3, *4-pin, 1.27mm semi-pitch Tyco Micromatch© connector*).

1 x USB 2.0 downlink/downstream (X4, *4-pin, 1.27mm semi-pitch Tyco Micromatch© connector*).

1 x RGB LED/Microphone/Sabotage connector *(CON2, 10-pin, 0.5mm pitch, FFC ZIF connector).*

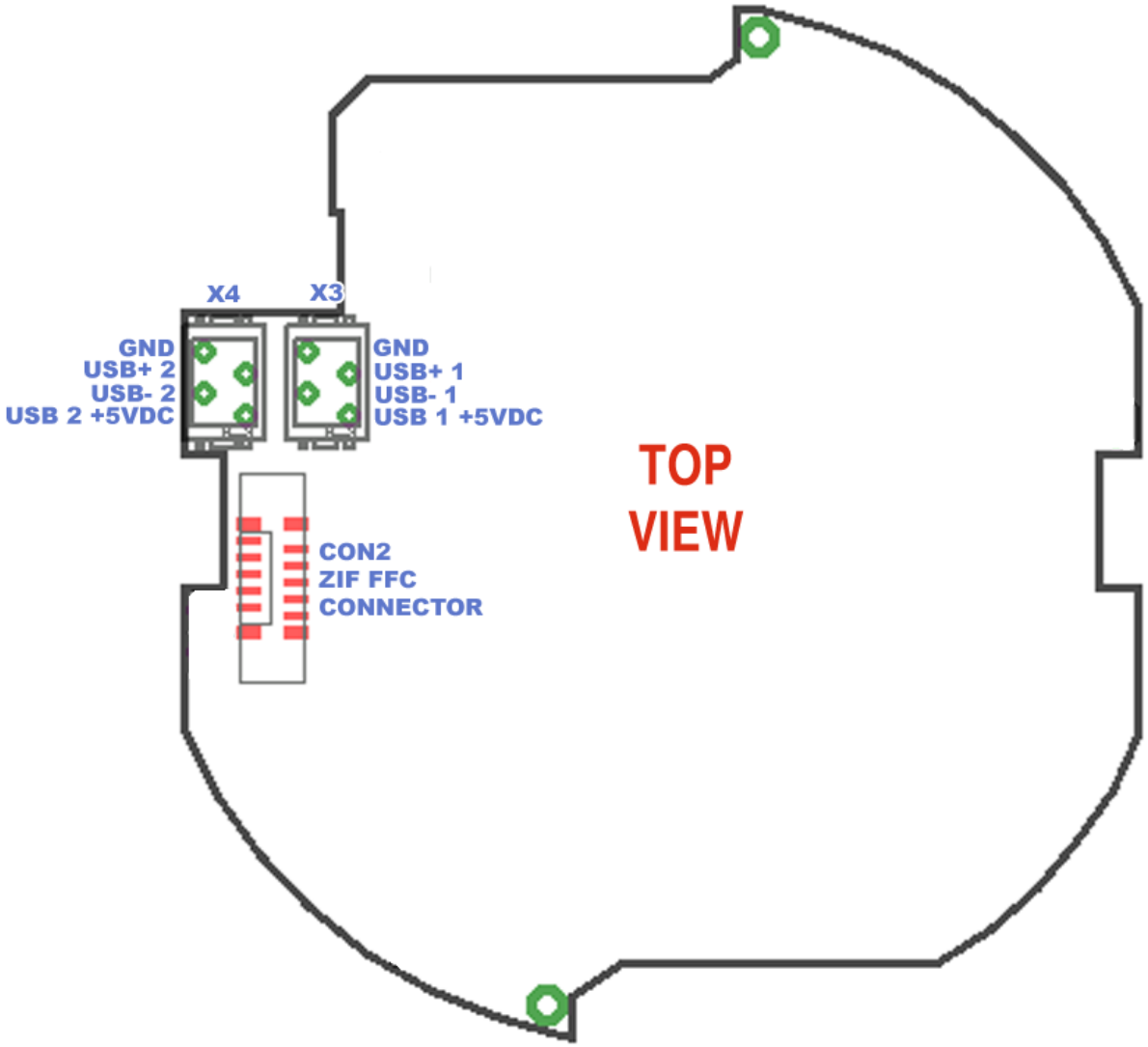

Image 24: Pin-out and localization of the PalmSecure-iO© external connectors (top).

6.7 Proprietary connectors (top side)

1 x Alternative sabotage switch connector *(JP4, 2-pin, 2.54mm pitch, single row, pin connector).*
1 x External speaker connector *(JP5, 2-pin, 2.54mm pitch, single row, pin connector).*
1 x External heater connector +5VDC/2.2A *(JP1, 2-pin, 2.54mm pitch, single row, pin connector).*

6.8 Jumpers (top side)

1 x Bootloader-update jumper (normally open) *(JP3, 2-pin, 2.54mm pitch, single row, pin connector).*

6.9 Optional LEDs (top side)

1 x CPU activity indication *(D5, bright blue).*
1 x Ethernet link indication *(D4, white, only on Ethernet version).*
1 x Ethernet speed indication *(D6, white, only on Ethernet version, on in 100MBit/s mode).*
1 x USB hub active indication *(D7, yellow).*
1 x USB hub speed indication *(D8, yellow, on when working in high-speed or 480MBit/s mode).*
1 x Serial communication TX activity *(D2, orange/red).*

1 x Serial communication RX activity *(D3, orange).*

6.10 Others (top side)

1 x PCB-mounted I²C/TWI temperature sensor *(IC10).*

1 x PCB-mounted 1-Wire silicon serial number *(IC15).*

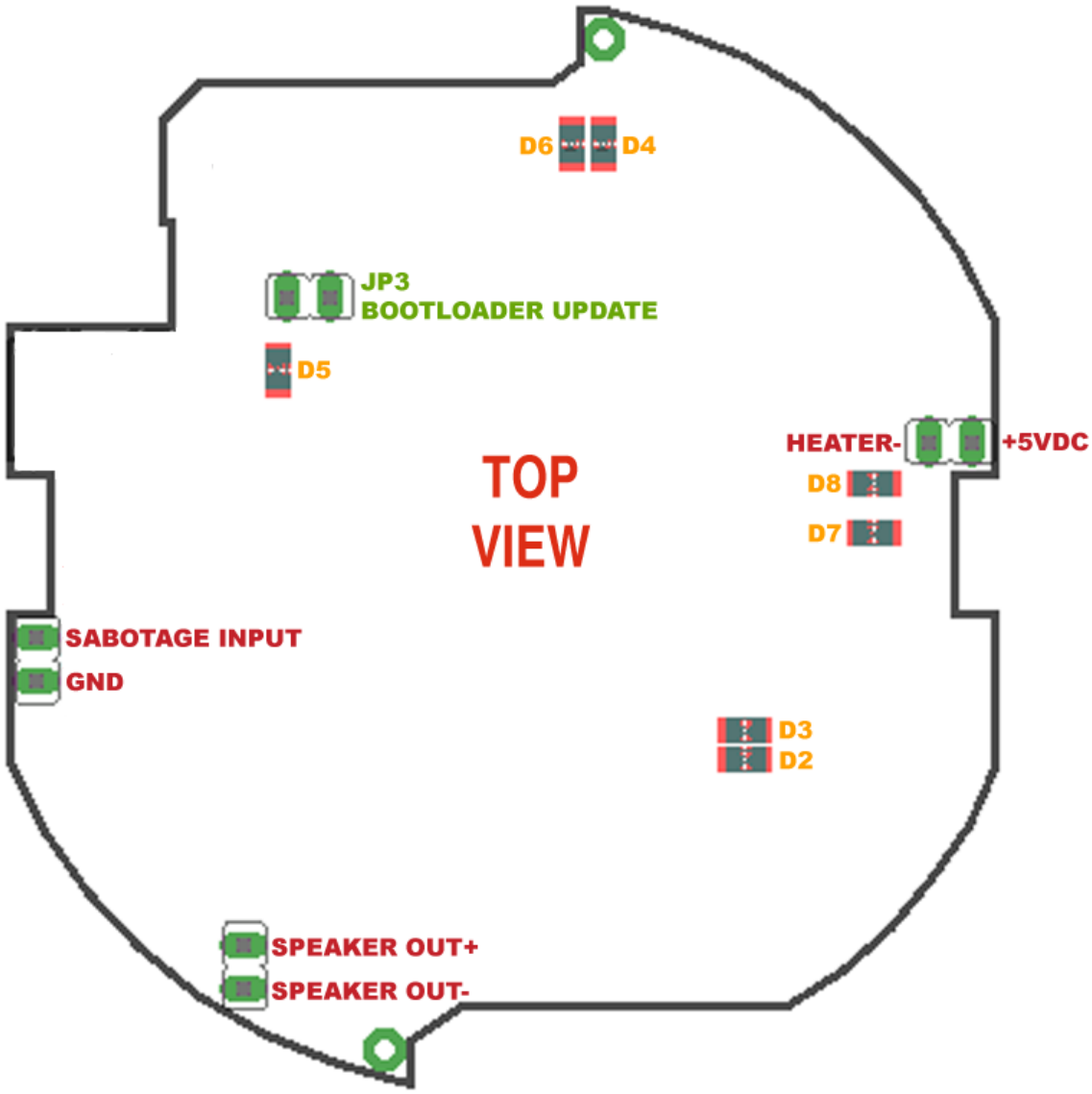

Image 25: Pin-out and localization of several PalmSecure-iO© connectors and components (top).

6.11 Installation connectors outdoor (top side)

1 x USB 2.0 uplink/upstream or 10/100BaseT *(ST3,5-pin, WAGO 218/2.5mm header).*

1 x Relay normally-open/normally-closed *(ST2,5-pin,WAGO 218/2.5mm header).*

2 x +50VDC tolerant/0.5A open-collector digital output *(ST2, 5-pin, WAGO 218/2.5mm header).*

4 x +24VDC tolerant Digital input *(ST8, 6-pin, WAGO 218/2.5mm header).*

1 x +5VDC tolerant Wiegand input *(ST11, 3-pin, WAGO 218/2.5mm header).*

1 x +5VDC tolerant Wiegand output *(ST9, 3-pin, WAGO 218/2.5mm header).*

1 x +5VDC CAN bus interface *(ST1, 3-pin, WAGO 218/2.5mm header).*

1 x RS-422/RS-485 or optional RS-232 interface *(ST14, 3-pin, WAGO 218/2.5mm header).*

1 x +5VDC tolerant SPI interface/Interrupt *(ST5, 5-pin, WAGO 218/2.5mm header).*

1 x +12VDC/0.5A Power supply *(ST12, 3-pin, WAGO 218/2.5mm header).*
1 x +5VDC/1.5A Help Power supply *(ST12, 3-pin, WAGO 218/2.5mm header).*
1 x +12VDC/25W Power Input for heater and help supply *(ST10, 6-pin, WAGO 218/2.5mm header).*
1 x Speaker output *(ST10, 6-pin, WAGO 218/2.5mm header).*
1 x USB 2.0 downlink/downstream *(ST7,5-pin, WAGO 218/2.5mm header).*
1 x +5VDC tolerant I2C interface/Interrupt *(ST4, 5-pin, WAGO 218/2.5mm header).*
1 x +5VDC tolerant 1-Wire interface *(ST5, 3-pin, WAGO 218/2.5mm header).*

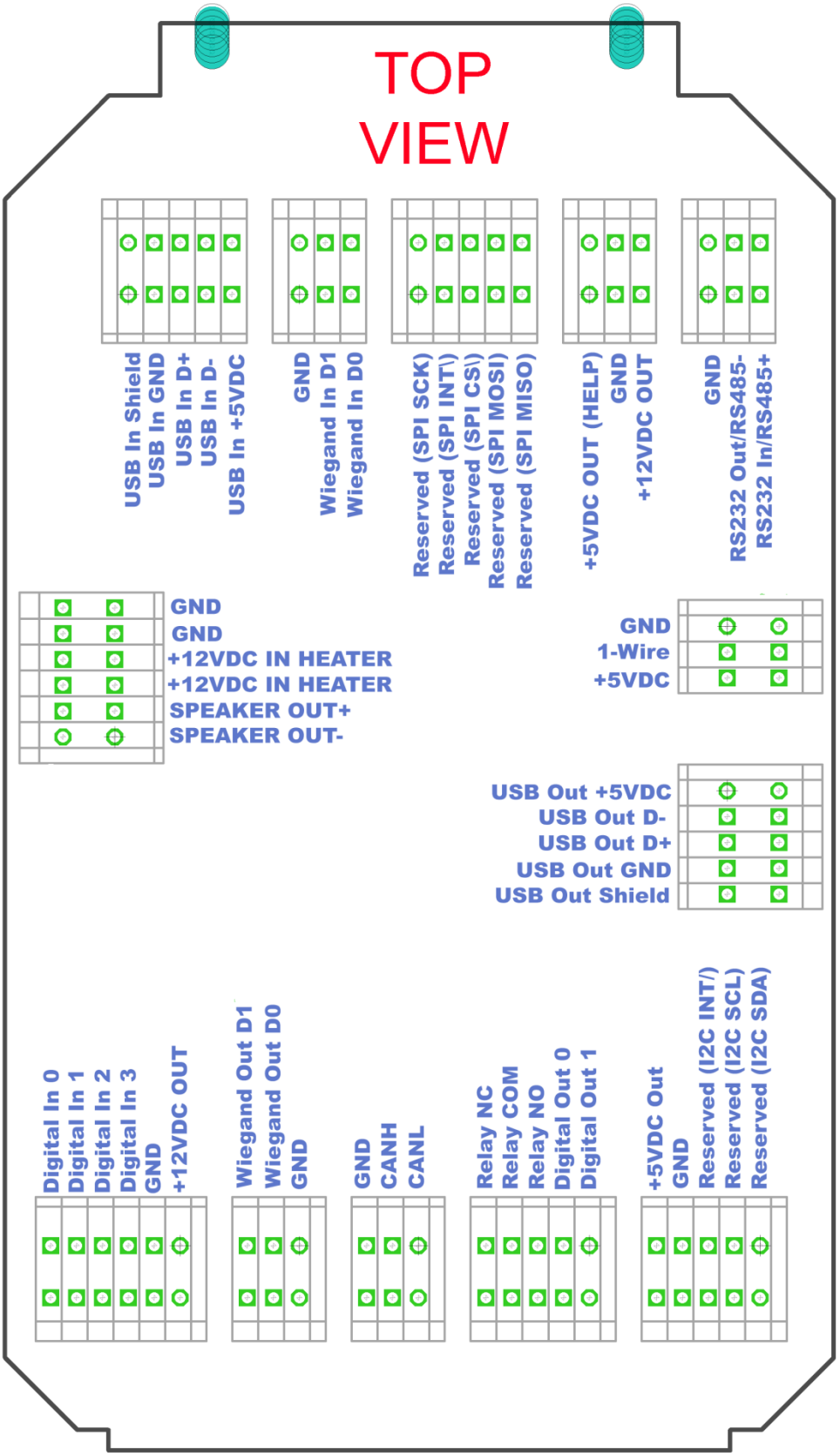

Image 26:Pin-out and localization of the PalmSecure-iO© outdoor external connectors (top).

6.12 Others outdoor (top side)

1 x CAN bus termination jumper (normally open) *(JP5, 2-pin, 2.54mm pitch, single row, pin connector).*

1 x RS-485 bus termination jumper (normally open) *(JP6, 2-pin, 2.54mm pitch, single row, pin connector).*

1 x Supply source selection jumper (normally 1-2) *(JP12, 3-pin, 2.54mm pitch, single row, pin connector).*
1 x PCB-mounted sabotage switch *(S1).*

Image 27: Pin-out and position of PalmSecure-iO© outdoor jumpers and components (top).

7. Electrical hardware installation

Basically there are several ways to connect the iCOGNIZE PalmSecure-iO© hardware platform to control the door strike and different other components like REX-button, door-open indicator, etc.

The latter, for example, can be connected to one of the four digital input ports available on the PalmSecure-iO© device. Changes on one or more of these input ports can be reported automatically to the upper layer device.

A door can be controlled i.e. opened in two different ways. The first and easiest is for demo purposes only due to very limited security. It uses the relay integrated in the PalmSecure-iO© and directly connects to a 12-24V DC or AC door strike. That way a very simple demonstration installation set-up is possible. The following images will explain that type of installation.

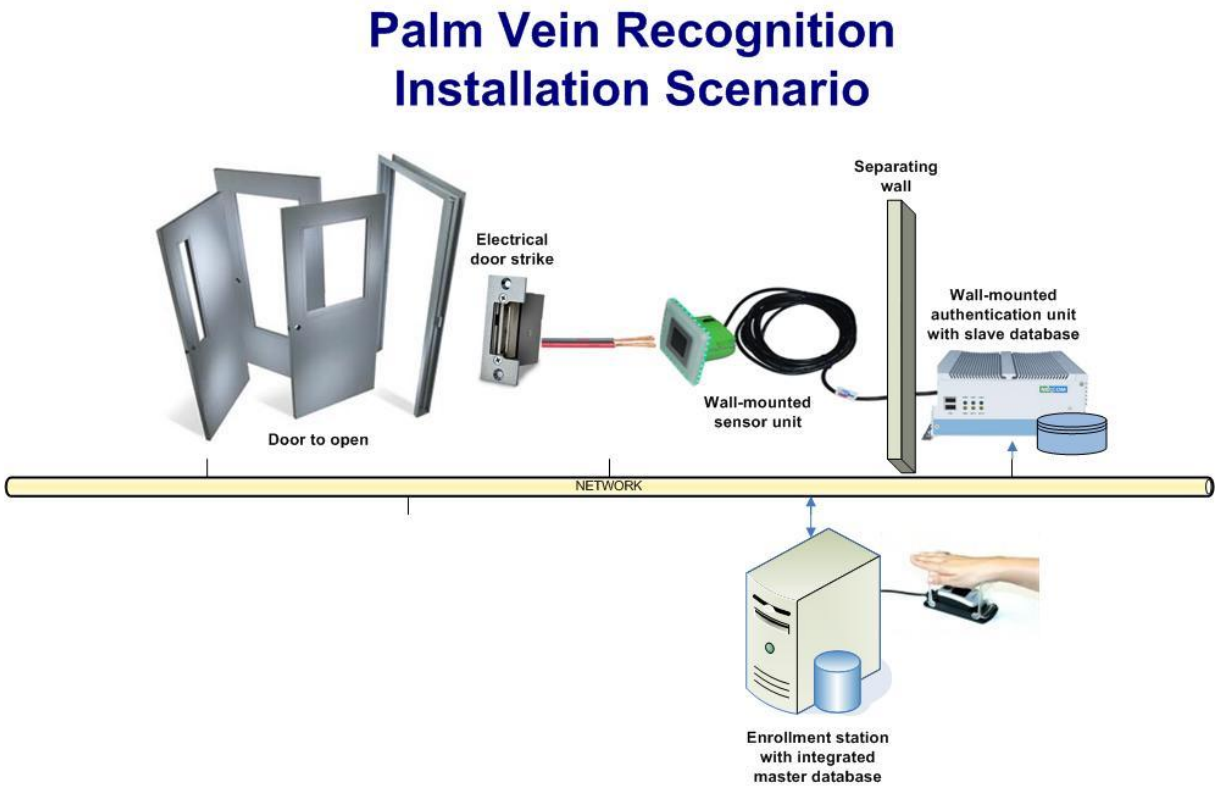

Image 28: iCOGNIZE PalmSecure-iO© demo installation scenario.

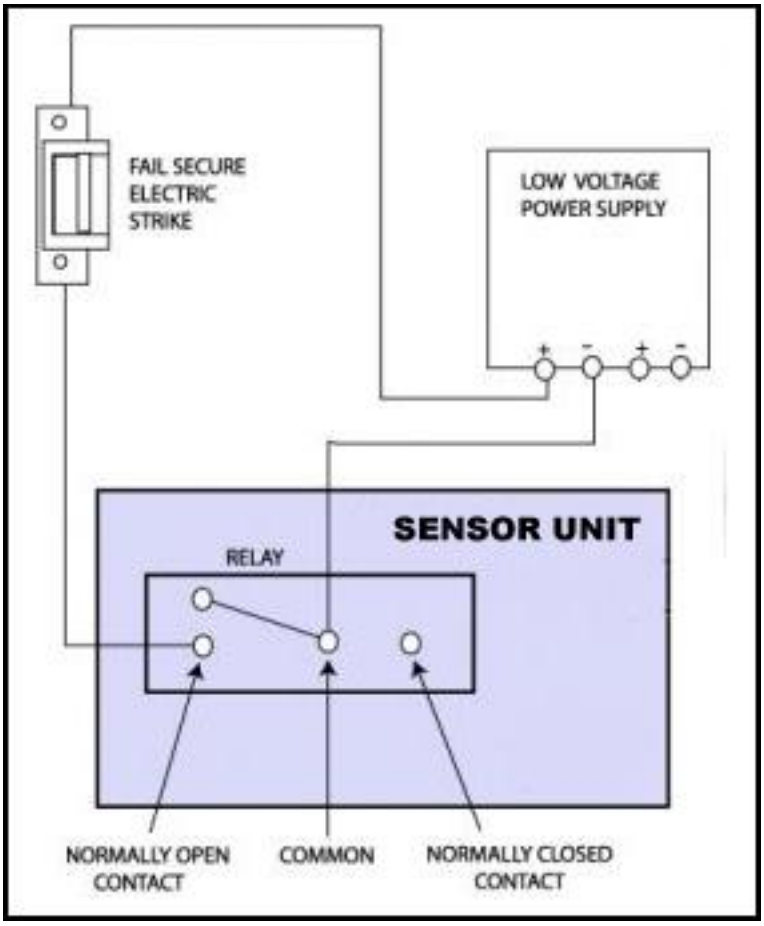

Image 29: Possible schematic for iCOGNIZE PalmSecure-iO© demo installation.

Alternatively the relay can be connected to an already installed door controller with a REX input for example.

A more secure and robust way to connect the platform to open a door for example is using a dedicated door controller receiving commands from the PalmSecure-iO© device by using one of the available hardware interfaces (Wiegand/ClockData, CAN, RS485, etc.). What interface will be finally used only depends on what the used door controller needs as input. That will be for sure the most common way to integrate the iCOGNIZE solution into existing or new installations.

Finally the different input interfaces available on the PalmSecure-iO© provide the functionality of multi-modal authentication. In that mode e.g. a RFID reader with one of the above mentioned inter-faces and connected to the PalmSecure-iO© delivers the red ID from the card to the iCOGNIZE Palm-Secure-iO© device. This in turn can forward the ID via the serial communication protocol to the authentication unit and triggers a verification process in-stead of an identification process. This may improve performance and security level of the complete system dramatically.

8. Mechanical hardware installation

The iCOGNIZE PalmSecure-iO© ManuScan© Indoor solution is prepared to fit into cavity-walls with wall-thickness between 10mm and 35mm and at least 75mm cavity depth. This situation one will find most often when doing installations in business buildings. Thus, be careful when planning to install into

concrete walls. Special preparation could be necessary to assemble the system in such a wall because the wall flush-mount may not fit or it could be impossible to affix it to the wall.
Furthermore, care should be taken for the best achievable air circulation because of the system power dissipation. This means, when installation is done into cavity-walls, remove probable existent mineral fibrous insulating material or others around the PalmSecure-iO©. In concrete walls heat conductivity is mostly good enough to remove heat from the device. Anyhow, no warranty can be given in that case. MTBF can be drastically reduced in that scenario.

8.1 Step-by-step mechanical installation overview

The following pictures will show the single steps necessary to install the iCOGNIZE PalmSecure-iO© into a cavity-wall and will most often take not more than 30 minutes if you follow exactly the advices.

Image 30: Drill a 68mm installation hole into the cavity-wall.

Image 31: Prepare the hole for installation (get USB and other cables off the wall, etc.).

Image 32: Connect controller unit to the cables and attach it into the hole.

Image 33: View from backside of the wall after proper installation.

9. Software installation

The PalmSecure-iO© provides different functionalities which, to be used, require some initial software preparations on the connected authentication device.

The main functionality is that of a USB 2.0 hub. It is realized by using an USB2514B chip from SMSC. Under a MS Windows© environment (XP, 2000, Vista, 7) all drivers required are provided by the OS and no further installation process is needed for that. Under other environments, the appropriate drivers must be downloaded from the manufacturer side and be installed former the employment.

The second functionality is the audio I/O. This is available thru a USB 2.0-to-Audio codec from Texas Instruments, the PCM2906B. Again, under a MS Windows© environment (XP, 2000, Vista, 7) all drivers required are provided by the OS and no further installation process is needed for that. After connecting the device, it will provide an additional soundcard and can be accessed exactly as such. Under other environments, the appropriate drivers must be downloaded from the manufacturer side and be installed former the employment.

Third there is the LED and I/O functionality. It uses a USB 2.0-to-RS232 converter chip (FT232R) from FTDI to provide a virtual COM port on the authentication unit. Therefore, under all environments, a driver has to be installed. It can be downloaded from the manufacturer homepage and has to be installed former the use of the device.

Additional drivers are required when using the Ethernet version. This is needed to provide virtual USB port functionality on the different platforms. Unfortunately currently only drivers for Microsoft Windows© platforms are available. Drivers for Linux will follow as soon as possible.

The Fujitsu PalmSecure© sensor brings its own drivers for the different environments to be installed.

Finally it could be possible to install further drivers, depending on what additional hardware is connected to the internal, spare USB 2.0 downlink.

10. Communication

Described here, one will find useful information for communication with PalmSecure-iO©. This will become necessary when writing third party software supporting the PalmSecure-iO©.

Remember that the device has an automatic host-fault or connection-loss detection. This means that the host has to send any of the commands below at least every 20 seconds, which is something like a heartbeat. If the device does not receive any command within 20 seconds, it will assume a connection break or host malfunction and will indicate that by quickly flashing the LEDs (red flashing when in Bootloader-operation, red-green flashing when in Application-operation). As soon as a command has been received again (except the "SETRGB" command) the old LEDs setting will recover.

After reset, device will start with flashing RGB LEDs to indicate operation, white flashing for Boot-loader-operation, green-orange flashing for Application-operation. Furthermore at least one beep should be heard on start-up.

When in flashing mode, RGB LEDs will flash white-red.

10.1 Interface settings

When accessing the PalmSecure-iO©, the communication flow will use a virtual COM port i.e. a serial communication scheme. Even if the port is not real, it provides different parameters, as a physical COM port do, that have to be set right to enable communication with the PalmSecure-iO©. This depends mainly on the setup internal to the platform CPU firmware and is set to the following values at system start-up

- **Baud rate:** *57600 Bps.*
- **Data bits:** *8 Bit.*
- **Stop bits:** *1 Bit.*
- **Parity:** *none.*
- **Flow control:** *none.*

Use these settings, for example on a PC, to enable PalmSecure-iO© communication with HyperTerminal, for example. That way, easy device testing is possible.

10.2 Temperature sensors

The PalmSecure-iO© hardware platform provides a couple of temperature sensors which can be red-out using the commands below. The table below shows which temperature sensor concerns to which component. Note that the 1-Wire sensors do not have a fixed order.

Temp. Sensor	Concerning to…
Board 1	Logic board temperature.
Board 2	Break-out board temperature.
Board 3	Reserved for future use.
1-Wire 1	Reserved for future use (if only one 1-Wire sensor is available, probably Fujitsu PalmSecure© sensor temperature).
1-Wire 2	Reserved for future use.
1-Wire 3	Reserved for future use.

10.3 Protocol

Another thing you have to take in account when communicating with the PalmSecure-iO© is what to send and how it has to be formatted. So that the device may understand what was meant with the command received. This will be described in the following two chapters.

10.3.1 Description

As already mentioned, the protocol used to control and read-out the PalmSecure-iO© is a plain-text human-readable ASCII command protocol, where every command traveling from the host device to the PalmSecure-iO© has the same typical format. This structure is very simple. Whenever sending a command or request, it will start with a keyword defining what to do in response to it. This keyword is followed by an equal sign and at least one numerical integer parameter indicating the consecutive number for that command. That way, an identical command or request sent different times can be ignored by the device. After that ID a number of different parameters may be transmitted, all separated by a comma. What type these parameters have and how many are expected depend on the command selected by the keyword. Some commands need a lot of parameters some do not request a single one (except the ID). Finally, the command will be closed by a carriage return (13Hex) and a new line character (10Hex). White spaces will be ignored by the device, the interpreter is working in case sensitive operation mode for the command ID. Parameters are also case-sensitive unless otherwise agreed in the following command list. The maximum command line buffer is limited to 512 bytes. Avoid sending more than this amount of data in a single command line. Violations may cause unexpected results or even make the device hanging.
After receiving a command formatted as defined above, the device will start processing the received command or request. First it will check the number of parameters received and the types of the parameters. If the count or types do not match, an error will be returned. This is just the string “ERROR” followed by the carriage return and new line character. This is true too, if the command execution failed because of another reason. If processing was successful, an acknowledge message depending on the command selected will be returned. Whatever this is, it will be terminated with a carriage return/new line sequence.
That is the same for asynchronous event messages, which are sent when an asynchronous event (like changes on the digital I/Os e.g.) appears and these asynchronous event notifications are enabled. The format of these notifications is the keyword “EVENT” followed by an equal sign, the number of the event triggering this notification and a carriage return/new line sequence (see example below).

Example:
EVENT=0\r\n

Meaning:
A device sabotage attempt has been detected.

Right now, there are several event numbers defined you can see in the table below.

Event ID	Description
0	Device sabotage has been detected.
1	A Wiegand/CLKDATA data packet has been received.
2	One or more levels on the general purpose digital inputs have changed.
3	A CAN bus data packet has been received.
4	A CAN bus error occurred.
5	A timeout triggered RS-485/RS-232 data packet has been received.
6	An end-of-data character triggered RS-485/RS-232 data packet has been received.
7	A G-shock event has been detected.
16	External temperature sensor 1 under-temperature alert (if connected).
17	External temperature sensor 2 under-temperature alert (if connected).
18	External temperature sensor 3 under-temperature alert (if connected).
32	External temperature sensor 1 over-temperature alert (if connected).
33	External temperature sensor 2 over-temperature alert (if connected).
34	External temperature sensor 3 over-temperature alert (if connected).

10.3.2 Commands

Following one will find the commands and request keywords the platform is able to execute, the parameters the appropriate command expects and what they will return.

Command string	Command category	Description	Acknowledge	Param. count	Example	Parameters
GETID	General device command/bootloader and app.	Returns the ID of that device which is "iCOGNIZE_IO_CONTROLLER" (for devices before or equal V2.x) or "iCOGNIZE_EMBEDDED_DEVICE" for newer devices. This can be used for auto-detection (open a COM port and send GETID command, if the answer is one of them above, it is an iCOGNIZE device). NOTE: it is perfect for auto-detection because this commands answers on every request, meaning it ignores the walking command index!	If it is an iCOGNIZE device, the device will answer with "iCOGNIZE_IO_CONTROLLER" (in case it is an old device) or with "iCOGNIZE_EMBEDDED_DEVICE" (for newer devices). *[Type: string].*	1	*Command:* GETID=1\r\n *Acknowledge:* iCOGNIZE_EMBEDDED_DEVICE\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
RESETINDEX	General device command/bootloader and app.	Resets the current walking index to its initial value of 1. This is helpful at the very beginning of device set-up or initialization when the user don't know what the current running index is. To provide that, the command ignores the walking command index.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* RESETINDEX=1\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETAPPNAME	General device command/bootloader and app.	Commands the device to return the current running software name. This is basically "APPLICATION" or "BOOTLOADER", depending on in which mode the device is currently running in. Normally the device will end-up in application mode after reset. Only when there is no executable application available (for example if former application flashing failed, etc.), the device will fall-back into bootloader mode to enable new device flashing.	The name of the current running software in the format of "APPNAME=$NAME$" where $NAME$ is going to be replaced by the actual running software name. Basically the software name will be "BOOTLOADER" or "APPLICATION". *[Type: string].*	1	*Command:* GETAPPNAME=2\r\n *Acknowledge:* APPNAME=APPLICATION\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETAPPDESC	General device command/bootloader and app.	Commands the device to return the current running software description. Basically this represents the device functionality in text form. The return value depends on the device connected and the here installed firmware.	The description to the current running software in the format of "APPDESC=$DESC$" where $DESC$ is going to be replaced by the actual running software description. *[Type: string].*	1	*Command:* GETAPPDESC=3\r\n *Acknowledge:* APPDESC=iCOGNIZE_TEST_DEVICE\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETAPPCOPYRIGHT	General device command/bootloader and app.	Commands the device to return the current running software copyright remark. The return value depends on the device connected and the here installed firmware.	The copyright informations to the current running software in the format of "APPCOPYRIGHT=$COPYRIGHT$" where $COPYRIGHT$ is going to be replaced by the actual running software copyright informations. *[Type: string].*	1	*Command:* GETAPPCOPYRIGHT=4\r\n *Acknowledge:* APPCOPYRIGHT=iCOGNIZE GmbH 2011\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETSERIAL	General device command/bootloader and app.	Commands the device to return it's serial number (which should be unique for every single device). The return value depends on the device connected (and obviously if there has been set an alternative serial number manually).	The serial number of the connected device in the format of "SERIAL=$SERIAL$" where $SERIAL$ is going to be replaced by the actual set device serial number. *[Type: string].*	1	*Command:* GETSERIAL=5\r\n *Acknowledge:* SERIAL=012424354355-124312125125\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETSERIAL	General device command/bootloader and app.	This command will set the device serial number to a custom user defined value. Any values are allowed, the number is stored as a string (maximum length: 128 bytes). NOTE: Due to flash ram access, executing this command may take some time (milliseconds).	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETSERIAL=6,ABCD-DEFG-0123\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The serial number to set as a string, maximum length 128 bytes. *[Type: string].*
CLRSERIAL	General device command/bootloader and app.	This command will clear the currently set device serial number. If the serial number is cleared and the device is reset, the serial number will be reset to its hardware defined serial number. NOTE: Due to flash ram access, executing this command may take some time (milliseconds).	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* CLRSERIAL=7\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETHWINFO	General device command/bootloader and app.	Commands the device to return the current running hardware informations. This depends on the device connected and contains the hardware type (depends on the hardware configuration like used CPU, etc.), the hardware kind (depends on the function the device provides) and the hardware revision number (depends on the production revision).	The hardware informations of the current connected device in the format of "HWINFO=%u1.%u2.%u3" where %u1 is going to be replaced by the actual hardware type number, %u2 with actual hardware kind number and %u3 with the actual hardware revision number. *[Type: three unsigned integers, separated by dots].*	1	*Command:* GETHWINFO=8\r\n *Acknowledge:* HWINFO=6.3.1\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

GETSWINFO	General device command/bootloader and app.	Commands the device to return the current running software informations. This depends on the device connected and the installed firmware. It contains the major version number, the minor version number, the revision number and the build count.	The software informations for the current connected device running firmware in the format of "SWINFO=%u1.%u2.%u3.%u4" where %u1 is going to be replaced by the actual major version number, %u2 with actual minor version number, %u3 with the actual software revision number and %u4 with the actual software build number. *[Type: four unsigned integers, separated by dots].*	1	*Command:* GETSWINFO=9\r\n *Acknowledge:* SWINFO=1.0.0.1\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETSWBUILDTYPE	General device command/bootloader and app.	Commands the device to return the current running software build type. This can contain "DEBUG" or "RELEASE" depending on the build type. It can further contain other build settings, for example "WATCHDOG DISABLED" or "WATCHDOG ENABLED". Normally the different build parameters are separated by slashes. For sure the return value depends on the connected device and the here installed firmware.	The software build type for the current connected device running firmware in the format of "SWBUILDTYPE=$BUILDTYPE$" where $BUILDTYPE$ is going to be replaced by the actual software build type. *[Type: string].*	1	*Command:* GETSWBUILDTYPE=10\r\n *Acknowledge:* SWBUILDTYPE=RELEASE/WATCHDOG ENABLED\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETSWBUILDTIME	General device command/bootloader and app.	Commands the device to return the time and date (ISO8601 format) when the currently running software has been built. The return value depends on the device connected and the here installed firmware.	The software build time and date for the current connected device running firmware in the format of "SWBUILDTIME=$BUILDTIME$" where $BUILDTIME$ is going to be replaced by the actual software build time and date in ISO8601 format. *[Type: string].*	1	*Command:* GETSWBUILDTIME=11\r\n *Acknowledge:* SWBUILDTIME=2010-12-24T00:00:00\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETCOMPILERVERSION	General device command/bootloader and app.	Commands the device to return the version of the compiler used to build the currently running software. The return value depends on the device connected and the here installed firmware.	The version of the compiler used to build the current connected device running firmware in the format of "COMPILERVERSION=%u1.%u2.%u3" where %u1 is going to be replaced by the used compiler major version number, %u2 by the used compiler minor version number and %u3 by the used compiler patch level number. *[Type: three unsigned integers, separated by dots].*	1	*Command:* GETCOMPILERVERSION=12\r\n *Acknowledge:* COMPILERVERSION=4.1.1\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETCOMPILERINFO	General device command/bootloader and app.	Commands the device to return informations concerning the compiler used to build the currently running software. The return value depends on the device connected and the here installed firmware.	The information to the compiler used to build the current connected device running firmware in the format of "COMPILERINFO=$COMPINFO$" where $COMPINFO$ is going to be replaced by the used compiler information string. *[Type: string].*	1	*Command:* GETCOMPILERINFO=13\r\n *Acknowledge:* COMPILERINFO=4.1.1\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETIAPBLOCKSIZE	General device command/bootloader and app.	Commands the device to return the size in bytes of one IAP (in-application programming) block. The running software is organized in flash ram in the manner of blocks, re-programmable and erasable independently. The size of one of such blocks can be requested that way, which is important when re-programming the device. The return value strongly depends on the connected device and the here running firmware.	The IAP block size of the currently connected device in the format of "IAPBLOCKSIZE=%u" where %u is going to be replaced by the actual device IAP block size. *[Type: unsigned integer].*	1	*Command:* GETIAPBLOCKSIZE=14\r\n *Acknowledge:* IAPBLOCKSIZE=2048\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETIAPBLOCKCOUNT	General device command/bootloader and app.	Commands the device to return the number of IAP (in-application programming) blocks available on device for firmware. The running software is organized in flash ram in the manner of blocks, re-programmable and erasable independently. The number of blocks can be requested that way, which is important when re-programming the device. This strongly depends on the connected device and the here running firmware.	The IAP block count of the currently connected device in the format of "IAPBLOCKCOUNT=%u" where %u is going to be replaced by the actual device IAP block count. *[Type: unsigned integer].*	1	*Command:* GETIAPBLOCKCOUNT=15\r\n *Acknowledge:* IAPBLOCKCOUNT=88\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
CALLBOOTLOADER	General device command/bootloader and app.	With that command, the device can be instructed to jump to the bootloader mode. This is important for device flashing, which can only be done in bootloader mode. If the device is already in bootloader mode or in application mode can be obtained using the "GETAPPNAME" command. If device is already in bootloader mode, the command will have no effect.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* CALLBOOTLOADER=16\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

CALLAPPLICATION	General device command/bootloader and app.	With that command, the device can be instructed to jump to the application mode. This is important after device flashing, to activate the application for example. The device will NOT jump back to application mode automatically after flashing. If the device is already in application mode or in bootloader mode can be obtained using the "GETAPPNAME" command. If the device is already in application mode, the command will have no effect.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* CALLAPPLICATION=17\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
ENABLEEVENTS	General device command/bootloader and app.	This command enables device asynchronous event notification. If this is not enabled, polling has to be used to check if inputs have changed, data has been arrived, temperature crossed limits, etc. After device reset or power-up this is NOT enabled by default.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* ENABLEEVENTS=17\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
DISABLEEVENTS	General device command/bootloader and app.	This command disables device asynchronous event notification. If this is disabled, polling has to be used to check if inputs have changed, data has been arrived, temperature crossed limits, etc. After device reset or power-up this is disabled by default.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* DISABLEEVENTS=18\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
ENABLELIFEDETECTION	General device command/bootloader and app.	This command enables device life detection notification. If this is enabled, LEDs will start flashing when no communication activity has been detected for a while (20secs). After device reset or power-up this is enabled by default.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* ENABLELIFEDETECTION=17\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
DISABLELIFEDETECTION	General device command/bootloader and app.	This command disables device life detection notification. If this is enabled, LEDs will start flashing when no communication activity has been detected for a while (20secs). After device reset or power-up this is enabled by default.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* DISABLELIFEDETECTION=18\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETEVENTSENABLE	General device command/bootloader and app.	Commands the device to return the current device asynchronous event notification status. This can also be useful to determine, if the device has been reset meanwhile. To do so, enable the asynchronous event notification at initialization and check status from time to time. If the device has been reset (because of malfunction, watchdog, etc.), the status will be set back to disabled. If one observes malfunction, please contact iCOGNIZE GmbH.	The current asynchronous event notification enable state of the connected device in the format of "GETEVENTSENABLE=$STATE$" where $STATE$ is going to be replaced by either "ENABLED" or "DISABLED", indicating if currently asynchronous event notification is enabled or not. *[Type: string].*	1	*Command:* GETEVENTSENABLE=19\r\n *Acknowledge:* GETEVENTSENABLE=ENABLED\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETWATCHDOGRST	General device command/bootloader and app.	Commands the device to return if the last reset reason has been the watchdog. Check that from time to time for reliable and proper function. If one observes watchdog resets, please contact iCOGNIZE GmbH.	The last reset reason of the connected device in the format of "GETWATCHDOGRST=$REASON$" where $REASON$ is going to be replaced by either "YES" or "NO", indicating if the last reset reason was the watchdog or not. *[Type: string].*	1	*Command:* GETWATCHDOGRST=20\r\n *Acknowledge:* GETWATCHDOGRST=NO\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETSCRATCHPAD	General device command/bootloader and app.	Commands the device to return the current scratchpad value. This provides no other functionality but is perfect to detect unpredictable or unexpected device resets. This is because the scratchpad register will be reset to a default value of 0 at power-up or reset. If some other value is written at the initialization sequence, this will be reset to zero again at device reset. Checking the scratchpad value from time to time will then indicate a reset if value changed to zero again. If one observes unpredicted resets, please contact iCOGNIZE GmbH.	The actual content of the scratchpad register of the connected device in the format of "GETSCRATCHPAD=%u" where %u is going to be replaced by the 32-bit value from the scratchpad register. *[Type: unsigned integer].*	1	*Command:* GETSCRATCHPAD=21\r\n *Acknowledge:* GETSCRATCHPAD=0\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETSCRATCHPAD	General device command/bootloader and app.	This command sets the scratchpad register value. This provides no other functionality but is perfect to detect unpredictable or unexpected device resets. This is because the scratchpad register will be reset to a default value of 0 at power-up or reset. If some other value is written at the initialization sequence, this will be reset to zero again at device reset. Checking the scratchpad value from time to time will then indicate a reset if value changed to zero again. If one observes unpredicted or unexpected resets, please contact iCOGNIZE GmbH.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETSCRATCHPAD=22,123\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The scratchpad value to set. *[Type: unsigned integer].*
SETSPEAKER	General device command/bootloader and app.	Commands the device to activate on board beeper. The command expects beep frequency in Hertz, the beep volume in percent and the duration of the beep in milliseconds. This is ideal for debugging or user feedback purposes.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	4	*Command:* SETSPEAKER=23,1000.0,100.0,1000.0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

						The beep frequency in Hertz. *[Type: float].*
						The beep volume in percent. *[Type: float].*
						The beep duration in milliseconds. *[Type: float].*
GETBOARDTEMP	General device command/bootloader and app.	Commands the device to return the value of a specified temperature sensor (in grad CSelsius). The sensors are all placed on appropriate device boards and thus represent the temperature of the device hardware itself. Normally up to 3 (depends on the connected device and its hardware and starting with index 1) different temperatures are available. To check the temperature of optional components, there is a possibility available to read-out 1-wire temperature sensors (see below).	The actual temperature of the board temperature sensor selected of the connected device in the format of "BOARDTEMP%u=%f" where %u is going to be replaced by the number of the selected temperature sensor and %f is going to be replaced by the measured temperature in grad Celsius. *[Type: unsigned integer, float].*	2	*Command:* GETBOARDTEMP=24,1\r\n *Acknowledge:* BOARDTEMP1=20.0\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The index of the temperature sensor to address (starting from 1). *[Type: unsigned integer, except zero].*
APPCHECK	General device command/bootloader and app.	That command requests the software to check if the currently flashed application part is basically enabled to run (using CRC checks and other verification methods). This might be helpful after flashing a new firmware. Thus it can be verified if flashing itself was successful in general.	The check result for the firmware of the connected device in the format of "APPCHECK=$RESULT$" where $RESULT$ is going to be replaced by either "OK" or "FAULT", indicating if the currently flashed application firmware is executable or not. *[Type: string].*	1	*Command:* APPCHECK=25\r\n *Acknowledge:* APPCHECK=OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
APPSIZE	General device command/bootloader and app.	Commands the device to return the actual size of the application part of the software. This is just for information purposes.	The total size of the application firmware of the connected device in the format of "APPSIZE=%u" where %u is going to be replaced by the flashed application firmware size in bytes. *[Type: unsigned integer].*	1	*Command:* APPSIZE=26\r\n *Acknowledge:* APPSIZE=66666\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
APPACTIVE	General device command/bootloader and app.	Commands the device to return if the currently flashed firmware has an activated application part. Activating an application is an important concept for reliability and safety, especially when flashing the device (see "ACTIVATEAPP" command for details).	The activation status of the application firmware of the connected device in the format of "APPACTIVE=$STATUS$" where $STATUS$ is going to be replaced by either "YES" or "NO", representing the current application firmware activation status. *[Type: string].*	1	*Command:* APPACTIVE=27\r\n *Acknowledge:* APPACTIVE=YES\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
RESETUSBRS232	General device command/bootloader and app.	The command advises the device to reset the USB-to-Serial converter used on the device. This command is normally never required and should only be used for testing cases. Anyway, because executing this command will break the device communication, acknowledge is sent BEFORE execution.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* RESETUSBRS232=28\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
RESETDEVICE	General device command/bootloader and app.	The command advises the device to reset itself. This command is normally never required and should only be used for testing cases or if malfunction is observed. In that case, please contact iCOGNIZE GmbH. Anyway, because executing this command will break the device communication, acknowledge is sent BEFORE execution.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* RESETDEVICE=29\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETBAUDRATE	General device command/bootloader and app.	This command sets the communication baud rate for the RS-232 interface. After reset or power-up, the device communicates with 57600 baud by default. This is for backward compatibility. When initializing the device, switching to 115200 baud (or even higher) is recommended to reduce communication overhead. When executing that command, the answer or response will be send in the old baud rate. After that, the device is switched to the new baudrate.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETBAUDRATE=30,115200\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The baud rate to set in baud. *[Type: unsigned integer, except zero].*

GETREPOSITORYID	General device command/bootloader and app.	Commands the device to return the current running software repository ID. Basically this represents the version control branch used for the installed firmware. The return value depends on the device connected and the here installed firmware. This is a value used for internal use and is more or less just for information purposes.	The ID of the version control repository from which the application firmware of the connected device was created in the format of "REPOSITORYID=ID" where ID is going to be replaced by used repository ID (where ID is actually Mercurial dependent). *[Type: string].*	1	*Command:* GETREPOSITORYID=31\r\n *Acknowledge:* REPOSITORYID=ABCDEFG\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
INITDOWNLOAD	General device command/bootloader only.	This command initiates the download of an application firmware block to be flashed when doing a firmware update. By calling that command the first time, the current application is going to be deactivated and the device switches to firmware update mode. Any other command than "EXITDOWNLOAD" or "DATADOWNLOAD" will force the device to leave the firmware update mode and leaving the device in a state, where only the bootloader will come up after reset or power-up. NOTE: Due to flash ram access, that command may take a while until responding execution (tenth or hundreds of milliseconds). Remember that this command is available in bootloader mode only! To finalize downloading correctly, at the end of the update process one has to initialize a data download using this command with the highest available IAP block number (accessible via "GETIAPBLOCKCOUNT" command). If there are no datas available for that last block, just close it using the "EXITDOWNLOAD" command. Thus, the application size is written to flash and the internal checksum will be calculated and stared in flash ram, too. This important to enable the application firmware to be activated at the end.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* INITDOWNLOAD=32,0,!IC(DOWNLOAD)+\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The flash block to initiate data downloading for, limited to the value returned from "GETIAPBLOCKCOUNT" command minus 1. *[Type: unsigned integer].*
						A magic cookie string to allow flash data download initiation, has to be "!IC(DOWNLOAD)+". *[Type: string].*
EXITDOWNLOAD	General device command/bootloader only.	This command finalizes a completely downloaded block of application firmware data or code. If the block was downloaded correctly (checksum checking is used to detect data corruption), flashing of the specified block will start on execution. The flashing result will be reported then. NOTE: Due to flash ram access, execution may take some time (tenth to hundreds of milliseconds). This command is available in bootloader mode only. Furthermore, the device has to be in firmware update mode, too. This is done by using the "INITDOWNLOAD" command before.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* EXITDOWNLOAD=33,12345\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

						The checksum for all the datas sent for the current block. This allows the device to verify if datas received equal the datas received as the device performs an internal checksum calculation, too. To calculate the CRC, every byte of data of the current block is given to a CRC calculation algorithm. This algorithm expects the CRC (16-bit value) of the former byte as a parameter, too. The algorithm has the following pseudo code:
						
						`tempCRC1 = inputDataByte ^ ( inputCRC & 0x0ff );`
						`tempCRC2 = ( tempCRC1 >> 1 ) ^ tempCRC1;`
						`outputCRC = ( ( WORD )( tempCRC2 >> 1 ) << 8 ) | ( inputCRC >> 8 );`
						`if ( tempCRC1 & 1 ) outputCRC ^= 0x040;`
						`if ( tempCRC2 & 1 ) outputCRC ^= 0x080;`
						`if ( crcCalcGetParity( tempCRC1 ) ) outputCRC ^= 0x0c001;`
						
						where **crcCalcGetParity** is:
						
						`inputByte = inputByte ^ ( inputByte >> 4 );`
						`inputByte = inputByte ^ ( inputByte >> 2 );`
						`inputByte = inputByte ^ ( inputByte >> 1 );`
						`outputParity = ( inputByte & 1 ) ? true : false;`
						
						[Type: unsigned integer].

DATADOWNLOAD	General device command/bootloader only.	This command sends (or downloads) a part of a firmware flash ram block, if the device firmware update mode has been entered before using the "INITDOWNLOAD" command. Checksum checking is used to detect downloaded data corruption. After all datas or code of a block has been downloaded, that specific block can be finalized using the "EXITDOWNLOAD" command. Because the communication input buffer of the device is not large enough to receive a complete flash block data chunk at once, data transfer has to be split into consecutive executions of this command, everyone with another bunch of data. To advise the device where in the flash block the following datas have to be placed, an address offset is used. First execution of this command after "INITDOWNLOAD" thus will start with an address offset of zero. If, for example, 16 bytes are transmitted with this call, next execution of that command will expect an address offset of 16 and a new bunch of data. This is repeated until all datas of a flash block have been transmitted.NOTE: This command is available in bootloader mode only. Furthermore, the device has to be in firmware update mode, too. This is done by using the "INITDOWNLOAD" command before. NOTE: Currently total command string length is limited to 512 bytes.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* DATADOWNLOAD=34,0,AB0123543BCFFE67D008943087DC4352FFEBA43FAD9B3249856490869 0865ABEFB43098BFD0754B093ABC2459084325 D00894359087DC4352FFEEBA43FAD9B3249856 4908690865AEFB430\r\n *Acknowledge:* OK\r\n Command: DATADOWNLOAD=35,75,AB0123543BCFFE67D008943087DC4352FFEBA43FAD9B3249856490869 90865ABEFB43098BFD0754B093ABC2459084325D00894359087DC4352FEEBA43FAD9B3249856 4908690865ABEFB430\r\n Acknowledge: OK\r\n . . .	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The address offset of the follwoing datas in flash data block. This value is limited to the value returned from "GETIAPBLOCKSIZE" command minus 1. Furthermore block address offset + number of bytes to transmit MUST NOT be larger than the value returned from "GETIAPBLOCKSIZE" command minus 1. Finally one has to make sure, that the offset is equal the formerly used offset + the number of datas transmitted with the last call of that command. Avoid sending command lines larger than 512 in total. Not doing so will cause the device to hang! *[Type: unsigned integer, except zero].*
						The datas to be transmitted to the device in the format of a string containing the datas in hexadecimal manner (ABCD0145, etc.). No hex-preamble or whatever is needed. Thus a byte will occupy 2 characters to be sent to the device. Valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. No lower case characters are allowed. *[Type: hexadecimal string].*
ACTIVATEAPP	General device command/application only.	This command activates the currently flashed application firmware. Activating means that the device directly jumps from the bootloader mode to the application mode after reset or power-up. When starting to flash a new application firmware, the application is automatically deactivated. Thus, the device prevents itself to end-up in a half-flashed or corrupted or non-working application after flashing. That would make the device unusable. So initiating an application firmware update process automatically forces the device to stay in bootloader after reset or power-up. When flashing is done correctly and the device reports an executable application (using the "APPCHECK" command), the device can be forced to execute the application firmware using the "CALLAPPLICATION" command. When the application firmware is working properly it will provide this command, meaning the application can only activate itself if it is running correctly. After executing that command with a properly and working application firmware, the device will come up with application mode after future reset- or power-up-cycles.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* ACTIVATEAPP=36\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETSABOTAGE	Device specific command/bootloader and app.	Command the device to return the current sabotage status. It will report if there has been a sabotage once and reset the sabotage status afterwards. So consecutive command execution will report no sabotage, if the reson has been removed. If asynchronous event notofication is enabled, this command has not to be polled. A sabotage will then reported via an event notification. Following that, this command can be executed to get the current state and to reset it.	The sabotage status of the connected device in the format of "SABOTAGE=$STATUS$" where $STATUS$ is going to be replaced by either "YES" or "NO", representing the current sabotage status. *[Type: string].*	1	*Command:* GETSABOTAGE=37\r\n *Acknowledge:* SABOTAGE=NO\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETDOUT	Device specific command/bootloader and app.	This command will set the available digital output ports on the device. The appropriate ports will change their states immediately after command execution. Here every bit in the output value is connected to a specific digital output.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETDOUT=38,3\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

						The value to be used setting the digital outputs on the device (32-bit wide). *[Type: unsigned integer].*
GETDOUT	Device specific command/bootloader and app.	Returns the actually set digital output value. This is a convenience command and should be used only rarely. Better let the upper layer application keep the current output status, so it will not get lost when the device shows malfunction or unexpected resets. Every bit in the red value is connected to a specific digital output port.	The actual digital output value of the connected device in the format of "DOUT=%u" where %u is going to be replaced by the actual digital output value representing the corresponding logic levels at the digital output ports. Basically 32 digital output ports are available, thus depending on the hardware not all are physically available at connectors. *[Type: unsigned integer].*	1	*Command:* GETDOUT=39\r\n *Acknowledge:* DOUT=3\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETDIN	Device specific command/bootloader and app.	Commands the device to return the current digital input states. Here again every bit in the returned value is connected to a specific digital input port. If asynchronous event notification is enabled, executing that command in a polling matter is not necessary. Only execute it once at the device initialization process to get the initial value. Every change on the digital input ports will then be reported via the asynchronous event notification. In reaction to that event, this command should then be executed to get the actual digital input states.	The actual digital input value of the connected device in the format of "DIN=%u" where %u is going to be replaced by the actual digital input value representing the corresponding logic levels at the digital input ports. Basically 32 digital input ports are available, thus depending on the hardware not all are physically available at connectors. *[Type: unsigned integer].*	1	*Command:* GETDIN=40\r\n *Acknowledge:* DIN=0\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETRELAIS	Device specific command/bootloader and app.	This sets the device relay output. The device is capable of only pulsing the relay without host interaction. Thus the command expects the pulsing status (pulse it on or off) and the pulse time in tenth of a second. If pulsing is not needed, use a pulse time of zero.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* SETRELAIS=41,1,10\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The pulse level required, which can be 0 or 1 meaning to pulse the relay off (0) or on (1). *[Type: boolean].*
						The pulse duration in tenth of a second. *[Type: unsigned integer].*
SETRGB	Device specific command/bootloader and app.	Commands the device to blink, fade and animate the 4 available RGB-LED banks. To save host processing power, a lot of LED animation stuff is done by the device itself. One can select which bank should be switched on. Furthermore a ramp start can be given and the lower and upper cycling limits. And that is true for every single color channel (red, green and blue). Finally the animation speed per channel and the animation mode (single cycle, re-cycle or ping-pong cycle) can be defined.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	17	*Command:* SETRGB=42,0,0,0,0,0,0,0,100,100,100,0.1,0.2,0.3,1,1,1,15\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The initial red LED intensity in percent. Values larger than 100 will be internally interpreted as 100%. Thus longer full on periods are achievable when cycling by setting values above 100. If this value is specified negative, initial intensity is going to be ignored and actual intensity is taken as initial cycle value. *[Type: float].*
						The initial green LED intensity in percent. Values larger than 100 will be internally interpreted as 100%. Thus longer full on periods are achievable when cycling by setting values above 100. If this value is specified negative, initial intensity is going to be ignored and actual intensity is taken as initial cycle value. *[Type: float].*
						The initial blue LED intensity in percent. Values larger than 100 will be internally interpreted as 100%. Thus longer full on periods are achievable when cycling by setting values above 100. If this value is specified negative, initial intensity is going to be ignored and actual intensity is taken as initial cycle value. *[Type: float].*
						The lower red LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*

The lower green LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*
The lower blue LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*
The upper red LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*
The upper green LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*
The upper blue LED cycling intensity in percent. Negative values will be internally interpreted as zero. Values larger than 100 will be internally interpreted as 100%. Thus longer full on or off periods are achievable when cycling by setting negative values or values above 100. *[Type: float].*
The red LED cycling speed as a percentage addend per internal tick, where a tick lasts 10ms. Thus an addend of 1 will ramp an initial value of 0 to full 100% in exactly 1sec. *[Type: float].*
The green LED cycling speed as a percentage addend per internal tick, where a tick lasts 10ms. Thus an addend of 1 will ramp an initial value of 0 to full 100% in exactly 1sec. *[Type: float].*
The blue LED cycling speed as a percentage addend per internal tick, where a tick lasts 10ms. Thus an addend of 1 will ramp an initial value of 0 to full 100% in exactly 1sec. *[Type: float].*
The red LED cycling mode. Currently 4 cycling modes are defined (starting from 0), more may come. **Mode 0:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, cycling will stop. Thus, this mode is ideal to ramp from a current unknown brightness setting to an end value by defining start and end intensity to the same value and choosing this mode. **Mode 1:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction and starts animating again until it reaches the upper intensity limit. Then cycling stops. Thus, this mode is kind of one-time-ping-pong animation. **Mode 2:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction

and starts animating again until it reaches the upper intensity limit. Then direction is changed again and process restarts. Thus, this mode is kind of infinite-ping-pong animation.
Mode 3:
Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, brightness is reset to the lower intensity limit and process restarts. Thus, this mode is kind of infinite-ramp animation.
[Type: unsigned integer].

The green LED cycling mode. Currently 4 cycling modes are defined (starting from 0), more may come.
Mode 0:
Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, cycling will stop. Thus, this mode is ideal to ramp from a current unknown brightness setting to an end value by defining start and end intensity to the same value and choosing this mode.
Mode 1:
Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction and starts animating again until it reaches the upper intensity limit. Then cycling stops. Thus, this mode is kind of one-time-ping-pong animation.
Mode 2:
Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction and starts animating again until it reaches the upper intensity limit. Then direction is changed again and process restarts. Thus, this mode is kind of infinite-ping-pong animation.
Mode 3:
Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, brightness is reset to the lower intensity limit and process restarts. Thus, this mode is kind of infinite-ramp animation.
[Type: unsigned integer].

						The blue LED cycling mode. Currently 4 cycling modes are defined (starting from 0), more may come. **Mode 0:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, cycling will stop. Thus, this mode is ideal to ramp from a current unknown brightness setting to an end value by defining start and end intensity to the same value and choosing this mode. **Mode 1:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction and starts animating again until it reaches the upper intensity limit. Then cycling stops. Thus, this mode is kind of one-time-ping-pong animation. **Mode 2:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If lower brightness limit is reached, cycling will change its direction and starts animating again until it reaches the upper intensity limit. Then direction is changed again and process restarts. Thus, this mode is kind of infinite-ping-pong animation. **Mode 3:** Cycling is started from initial brightness or current brightness, depending on the value given as initial value. If current or initial brightness in combination with given brightness cycling addend would run out of bounds, direction of cycling is going to be adjusted. If upper brightness limit is reached, brightness is reset to the lower intensity limit and process restarts. Thus, this mode is kind of infinite-ramp animation. *[Type: unsigned integer].*
						A mask value enabling the 4 different LED banks available, where every single of the first 4 bits is dedicated to a specific bank. Thus the value range is from 0 to 15, where 0 disables all banks and 15 enables all banks. *[Type: unsigned integer].*
SETUSBPOWER	Device specific command/bootloader and app.	This command switches the +5V supply power on the external USB ports on or off. This is pretty important for several devices, specially the Fujitsu PalmSecure sensor. Sometimes that sensor seems to run in a dead-lock from where it only can be "re-animated" by a power on-off-on cycle. Here this command is very helpful. Again every single external USB port is connected to a special bit in the power control value given with that command. Furthermore this command needs to be used after power-on or reset as the USB power is initially switched off. If, for example, the PalmSecure sensor temperature is below 0°C, the external USB port power is switched off by default to prevent sensor damage. The power has then to be enabled manually using this command. Normally one would switch on the heater then, wait for the temperature raising above 0°C and enable the external USB port power afterwards. NOTE: Notice that in normal operation the external USB port power will NOT automatically switch off when environment i.e. PalmSecure sensor temperature drops below 0°C. During operation the external control software has to maintain that!	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETUSBPOWER=43,3\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The value to be used setting the external USB port power on the device (32-bit wide). *[Type: unsigned integer].*

GETUSBPOWER	Device specific command/bootloader and app.	Advises the device to returns the current external USB port power state. This might be helpful to check-out the current state of the USB power after power-up or reset. Because, as mentioned above, power could be disabled on reset or power-up when environment is too cold.	The actual external USB ports power enable value of the connected device in the format of "USBPOWER=%u" where %u is going to be replaced by the actual external USB ports power enable value representing the corresponding power states at every single external USB port. Basically 32 external USB ports are available, thus depending on the hardware not all are physically available at connectors. [Type: unsigned integer].	1	*Command:* GETUSBPOWER=44\r\n *Acknowledge:* USBPOWER=3\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SWITCHHEATER	Device specific command/bootloader and app.	With that command the device heater con be switched on or off. Normally, when talking about the indoor version of the palm vein recognition system, this command has never to be used as the system temperature should never drop below 0°C.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	2	*Command:* SETHEATER=45,1\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The heater switching state, which can be 0 or 1 meaning to switch the heater off (0) or on (1). *[Type: boolean].*
GETHEATER	Device specific command/bootloader and app.	Hereby the device is commanded to return the current status of the system heater. It is to determine if the heater is currently switched on or off. If operation temperature is in valid boundaries, the heater should be switched of because of its enormous power consumption. For the indoor version of the device, this command is probably of less interest.	The current heater state of the connected device in the format of "HEATER=$STATUS$" where $STATUS$ is going to be replaced by either "ON" or "OFF", representing the heater switch status. *[Type: string].*	1	*Command:* GETHEATER=46\r\n *Acknowledge:* HEATER=ON\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GET1WIRETEMP	Device specific command/bootloader and app.	Commands the device to return the value of a specified external temperature sensor (in grad Celsius). The sensors are all placed on external components and thus represent the temperature of the device extensions. Normally up to 3 (depends on the connected device and its hardware and starting with index 1) different temperatures are available. Basically all temperature sensors should report proper and valid temperatures before the complete system can be powered-up (external USB port power for example). The temperatures returned here are the more important system parameters. Board temperatures are less important as the device hardware itself is constructed to work in a wide temperature range (-45°C up to 85°C) where external components (like the PalmSecure sensor for example) are not. Therefore upper layer control software should take care of these operation parameters. To reduce polling activity for these values, asynchronous event notification can be configured using the following two commands.	The actual temperature of the external temperature sensor selected of the connected device in the format of "1WIRETEMP%u=%f" where %u is going to be replaced by the number of the selected temperature sensor and %f is going to be replaced by the measured temperature in grad Celsius. *[Type: unsigned integer, float].*	2	*Command:* GET1WIRETEMP=47,1\r\n *Acknowledge:* 1WIRETEMP1=25.0\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The index of the temperature sensor to address (starting from 1). *[Type: unsigned integer, except zero].*
SET1WIRETEMPLOWERLIMIT	Device specific command/bootloader and app.	This command sets the lower temperature limit for a specific external component temperature or the temperature delivered by an external component 1-wire temperature sensor. As long as the temperature of the specified sensor lies below the configured limit, the device will drop asynchronous event notification messages (in case asynchronous event notification is enabled).	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* SET1WIRETEMPLOWERLIMIT=48,1,-1.0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The index of the temperature sensor to address (starting from 1). *[Type: unsigned integer, except zero].*
						The lower temperature limit where an asynchronous event notification is going to be triggered, when specified sensor temperature drops below. *[Type: float].*
SET1WIRETEMPUPPERLIMIT	Device specific command/bootloader and app.	This command sets the upper temperature limit for a specific external component temperature or the temperature delivered by an external component 1-wire temperature sensor. As long as the temperature of the specified sensor lies above the configured limit, the device will drop asynchronous event notification messages (in case asynchronous event notification is enabled).	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* SET1WIRETEMPUPPERLIMIT=49,1,60.0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The index of the temperature sensor to address (starting from 1). *[Type: unsigned integer, except zero].*

						The upper temperature limit where an asynchronous event notification is going to be triggered, when specified sensor temperature raises above. *[Type: float].*
WIEGANDOUT	Device specific command/application only.	This command is used to initiate Wiegand data transmission using the Wiegand/CLKDATA output interface. Specific parameters define the properties of this single transmission as pulse width (in microseconds), pause time (in microseconds) and the bit count. For sure that datas to be transmitted have to be specified, too. MSB of the specified data will be sent out first. Pay attention to send at least as much data bits as defined by the bit count number. Extra data bits will be cut.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	5	*Command:* WIEGANDOUT=50,26,AB56BA5,150,1000\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The number of bits to be transmitted. Currently the number of bits is limited to 128. Zero is not allowed. *[Type: unsigned integer].*
						The datas to be transmitted over the Wiegand/CLKDATA interface of the device in the format of a string containing the datas in hexadecimal manner (ABCD0145, etc.). No hex-preamble or whatever is needed. Thus a byte will occupy 2 characters to be sent to the device. Valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E and F. Upper and lower case characters are allowed. Maximum data size is 16 byte, resulting in a maximum of 128 bits. The most significant bit is going to be sent first. *[Type: hexadecimal string].*
						The Wiegand interface pulse time in microseconds. *[Type: unsigned integer].*
						The Wiegand interface pause time in microseconds. *[Type: unsigned integer].*
CLKDATAOUT	Device specific command/application only.	This command is used to initiate CLKDATA data transmission using the Wiegand/CLKDATA output interface. Specific parameters define the properties of this single transmission as period time (in microseconds), clock and data port polarity, clock/data port swapping and the bit count. For sure that datas to be transmitted have to be specified, too. MSB of the specified data will be sent out first. Pay attention to send at least as much data bits as defined by the bit count number. Extra data bits will be cut.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	7	*Command:* CLKDATAOUT=51,26,AB56BA5,100,0,0,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The number of bits to be transmitted over the Wiegand/CLKDATA output interface. Currently the number of bits is limited to 128. Zero is not allowed. *[Type: unsigned integer].*
						The datas to be transmitted over the Wiegand/CLKDATA output interface of the device in the format of a string containing the datas in hexadecimal manner (ABCD0145, etc.). No hex-preamble or whatever is needed. Thus a byte will occupy 2 characters to be sent to the device. Valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E and F. Upper and lower case characters are allowed. Maximum data size is 16 byte, resulting in a maximum of 128 bits. The most significant bit is going to be sent first. *[Type: hexadecimal string].*
						The CLD/DATA output interface period time in microseconds. *[Type: unsigned integer].*
						A flag, if standard CLK and DATA output port pin-out should be swapped. Means defining a 0 here leaves the pin-out in standard configuration, defining a 1 will swap the standard pin-out on the device. *[Type: boolean].*
						A flag, if standard CLK output port polarity should be inverted. Means defining a 0 here leaves the CLK output port polarity in standard configuration (active low), defining a 1 will invert the standard CLK output port polarity (active high). *[Type: boolean].*

						A flag, if standard DATA output port polarity should be inverted. Means defining a 0 here leaves the DATA output port polarity in standard configuration (active high), defining a 1 will invert the standard DATA output port polarity (active low). *[Type: boolean].*
GETWIEGANDIN	Device specific command/application only.	This command returns the last received bunch of data from the Wiegand/CLKDATA input interface. The data packets are marked with indices to determine if the datas red-back are new datas or prior ones. In general, one does not have to poll that command permanently to watch out for new datas. In addition to receiving datas an asynchronous event notification is dropped to inform the upper layer control software that new data has been arrived (in case asynchronous event notification is enabled). The datas available may depend from the mode the Wiegand/CLKDATA input interface is currently working in (Wiegand or CLKDATA mode). This can be set using the "ENABLEWIEGANDIN" or "ENABLECLKDATAIN" commands. Be aware that these commands may set additional interface operation parameters! Wiegand is enabled by default after power-on or reset. The first arriving bit will be output as MSB.	The actual available Wiegand/CLKDATA datas of the connected device in the format of "WIEGAND_INPUT=%u1,%u2,$DATA$" where %u1 is going to be replaced by the actual walking index of received Wiegand/CLKDATA message (starting from 0), %u2 by the last received number of bits and $DATA$ by the datas received in hexadecimal style. Care should be taken when calling that command and no message has been received until now. Then the returned acknowledge will be "WIEGAND_INPUT=NONE". This has been implemented for backward compatibility. *[Type: unsigned integer, unsigned integer, hex. string].*	1	*Command:* GETWIEGANDIN=52\r\n *Acknowledge:* WIEGAND_INPUT=1,26,AABCDEF\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
RESETUSBHUB	Device specific command/application only.	The command advises the device to reset the internal USB hub used on the device. This command is normally never required and should only be used for testing cases. Anyway, because executing this command will break the device communication, acknowledge is sent BEFORE execution.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* RESETUSBHUB=53\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
RESETUSBSERVER	Device specific command/application only.	The command advises the device to reset the internal USB-to-Ethernet server used on the device. This command is normally never required and should only be used for testing cases. Anyway, because executing this command will break the device communication, acknowledge is sent BEFORE execution.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* RESETUSBSERVER=54\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETCANCONF	Device specific command/application only.	This command sets the global CAN bus interface operation parameters. These are important for reception AND transmission. These parameters are the bit rate in Hertz, the number of sub-sample clock ticks per period, the sub-sampling sample start clock tick number, the triple-sampling enable flag, the period stretch sub-sampling clock tick count and a test-enable flag. One should remember to configure the interface properly after reset or power-up if it is going to be used. Even if not, proper configuration with default parameters is recommended to avoid unexpected behavior.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	7	*Command:* SETCANCONF=55,1000000,8,6,1,2,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The baud rate to be used on CAN bus. *[Type: unsigned integer, except zero].*
						The number of sub-sampling clock ticks per CAN bus clock period. It is important to split the CAN bus period into smaller sub-sample pieces. Thus the sampling point of the CAN bus level can be specified in finer granularity. See external documents for detailed description, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example. *[Type: unsigned integer].*
						The number of the sub-sampling clock ticks where CAN bus logic level sampling should appear. Thus the sampling point of the CAN bus level can be specified in finer granularity and shifted to the point where physical bus installation delivers best results concerning reliability and bus integrity. See external documents for detailed description of CAN bus, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example. *[Type: unsigned integer].*
						A flag, if CAN bus logic level triple sampling is enabled or not. Means defining a 0 here disables the triple-sampling feature, defining a 1 will enable the triple-sampling feature. For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: boolean].*

						The maximum number of CAN bus period sub-sampling clock ticks a CAN bit period can be shortened or lengthened for re-synchronization. This is sometime also referred as the synchronization jump width. See external documents for detailed description of CAN bus, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example. *[Type: unsigned integer].*
						A flag, if CAN bus self-testing is enabled or not. Means defining a 0 here disables the self-testing feature, defining a 1 will enable self-testing feature. Basically it disables the internal check, if a packet has been acknowledged on the bus, what can only be done by a second device on the CAN bus. Normally, if a packet was not acknowledged on the bus it is intended to be invalid and thus silently discarded. Enabling the self-test feature, this check is switched off and that way tests can be performed even if there is only a single device on the bus. Remember that when using the local-loop-back functionality in the "CANOUT" command. The device using the "CANOUT" command with local-loop-back will only receive its own message if either self-test feature is enabled or it is not the only device on the CAN bus (where the foreign device has to acknowledge the sent packet). For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: boolean].*
GETCANTXREADY	Device specific command/application only.	This command is used to check, if the CAN transmission buffer is ready to accept new datas to be transmitted. It should be used before using the CANOUT command to prevent data loss. Especially on fast consecutive writes i.e. CAN bus send actions it is important to use that command to check system for CAN bus transmission readiness.	The current CAN bus transmit buffer ready status of the connected device in the format of "CANTXREADY=$STATUS$" where $STATUS$ is going to be replaced by either "YES" or "NO", representing the current CAN bus transmit buffer ready status. *[Type: string].*	1	*Command:* GETCANTXREADY=56\r\n *Acknowledge:* CANTXREADY=YES\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
CANOUT	Device specific command/application only.	This command is used to initiate CAN bus data transmission using the CAN bus interface. Specific parameters define the properties of this single transmission as there are flag for extended frame, flag for remote frame, the frame identifier, the number of bytes to transmit and a loop-back enable flag. For sure that datas to be transmitted have to be specified, too. Global parameters like baud rate etc. are taken from the values formerly set by the "SETCANCONF" command. So prior to using this command, CAN bus interface should be configured properly using the "SETCANCONF" command.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	7	*Command:* CANOUT=57,1,0,12345,1032547698EFCDBA,8,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						A flag, if CAN bus frame to send is of extended style or not. If this is 0, the next transmit CAN bus message will be sent with an 11 bit identifier, while if it's 1, the message will be sent with a 29 bit identifier. For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: boolean].*
						A flag, if CAN bus frame to send is of remote style or not. This value is sent in the RTR bit of the next transmit CAN bus message. If this bit is 0, the number of data bytes called out by the DLC field is sent. If it's 1, a remote frame is sent, containing a request for that number of bytes. For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: boolean].*
						The CAN bus message identifier to send. For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: unsigned integer].*

						The datas to be transmitted over the CAN bus interface of the device in the format of a string containing the datas in hexadecimal manner (ABCD0145, etc.). No hex-preamble or whatever is needed. Thus a byte will occupy 2 characters to be sent to the device. Valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E and F. Upper and lower case characters are allowed. Maximum data size is 8 byte. *[Type: hexadecimal string].*
						The number of bytes to send in the CAN bus message or the number of bytes requested (if it is an remote style frame). The maximum number is 8. For further information consult external documents, "UM10114: LPC21xx and LPC22xx User manual" from NXP for example (specially the CAN bus part). *[Type: unsigned integer].*
						A flag, if local-loop CAN bus communication is enabled. Means defining a 1 will enable the CAN bus interface to receive its own sent datas, defining a 0 witches that feature off. This is ideal for testing purposes and should normally be switched off in standard operation. When the sending device is the only device on CAN bus, additionally the self-test feature has to be enabled to allow the device to receive its own sent datas. See the "SETCANCONF" command for further details. *[Type: boolean].*
GETCANIN	Device specific command/application only.	This command returns the last received bunch of data from the CAN bus interface. The data packets are marked with indices to determine if the datas red-back are new datas or prior ones. In general, one does not have to poll that command permanently to watch out for new datas. In addition to receiving datas an asynchronous event notification is dropped to inform the upper layer control software that new data has been arrived (in case asynchronous event notification is enabled). The general interface operation parameters are set using the "SETCANCONF" command. Thus the "SETCANCONF" command should be called at the initialization of the device to ensure proper interface operation.	The actual available CAN bus datas of the connected device in the format of "CAN_INPUT=%u1,%u2,%u3,$DATA$" where %u1 is going to be replaced by the actual walking index of received CAN bus message (starting from 0), %u2 by the CAN bus message status, %u3 by the CAN bus message identifier and $DATA$ by the datas received in hexadecimal style. *[Type: unsigned integer, unsigned integer, unsigned integer, hex. string].*	1	*Command:* GETCANIN=58\r\n *Acknowledge:* CAN_INPUT=1,898989,12345,1032547698EFCDBA\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
ENABLEWIEGANDIN	Device specific command/application only.	This command switches the Wiegand/CLKDATA input interface of the device to the Wiegand mode. That way the device will react on Wiegand messages received on the specific interface and interpret the levels and timings correctly to store the received datas in internal buffer. Working in that mode, the device will NOT receive or interpret CLK/DATA messages on the Wiegand/CLKDATA input interface correctly. That mode is activated by default after reset or power-up.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	1	*Command:* ENABLEWIEGANDIN=59\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
ENABLECLKDATAIN	Device specific command/application only.	This command switches the Wiegand/CLKDATA input interface of the device to the CLK/DATA mode. That way the device will react on CLK/DATA messages received on the specific interface and interpret the levels and timings correctly to store the received datas in internal buffer. To fully provide that functionality, additional parameters are sent with that command. These are: clock polarity and clock port/data port swapping. Working in that mode, the device will NOT receive or interpret Wiegand messages on the Wiegand/CLKDATA input interface correctly. That mode is disabled by default after reset or power-up.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* ENABLECLKDATAIN=60,0,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						A flag, if level sampling on DATA input port should take place on the rising edge of the CLK input port or at the falling edge. Means defining a 0 here samples the DATA input port level on the falling edge of the CLK input port, defining a 1 will enable level sampling on the rising edge of the CLK input port. *[Type: boolean].*
						A flag, if standard CLK and DATA output port pin-out should be swapped. Means defining a 0 here leaves the pin-out in standard configuration, defining a 1 will swap the standard pin-out on the device. *[Type: boolean].*

SETRS485CONF	Device specific command/application only.	This command sets the global RS-485/RS-232 interface operation parameters. These are important for reception AND transmission. These parameters are the baud rate, a parity-enable flag, a sticky-parity-enable flag, an even-parity-enable flag, the data bit count, the stop bit count, a reception timeout time in milliseconds and an end-of-packet marker. One should remember to configure the interface properly after reset or power-up if it is going to be used. Even if not, proper configuration with default parameters is recommended to avoid unexpected behavior.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	9	*Command:* SETRS485CONF=61,1000000,0,0,0,8,1,5,-1,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). *[Type: unsigned integer, except zero].*
						The baud rate to be used on RS-485/RS-232 interface. *[Type: unsigned integer, except zero].*
						A flag, if parity handling is enabled on RS-485/RS-232 interface. Means defining a 0 here disables parity handling, defining a 1 enables parity handling. *[Type: boolean].*
						A flag, if sticky parity handling is enabled on RS-485/RS-232 interface. Means defining a 0 here disables sticky parity handling, defining a 1 enables sticky parity handling. Enabling sticky parity without enabling normal parity (see former parameter) is without effect. *[Type: boolean].*
						A flag, if parity handling uses odd or even parity on RS-485/RS-232 interface. Means defining a 0 here enables odd parity handling, defining a 1 enables even parity handling. Defining odd or even parity handling without enabling normal parity (see former parameters) is without effect. *[Type: boolean].*
						The number of data bits used for transmission and reception on RS-485/RS-232 interface. Valid values are 5, 6, 7 and 8. *[Type: unsigned integer].*
						The number of stop bits used for transmission and reception on RS-485/RS-232 interface. Valid values are 0 and 1. *[Type: unsigned integer].*
						The reception timeout time in milliseconds. Defines the time to wait after a character has been received before an asynchronous event notification is going to be triggered (if it was formerly enabled using the "ENABLEEVENTS" command). If a new character has been received within the waiting period, the timer is going to be reset. Using that, a packet separation can be performed if the packets do not carry a special end-of-packet character at the end of a data block and there is a transmission gap of a minimum time between consecutive packets on the bus. If timeout time is specified to zero, reception timeout asynchronous event notification is disabled. *[Type: unsigned integer].*
						This defines the end-of-packet marker. If this character is received, it will be interpreted as an end-of-packet notification and an asynchronous event notification is going to be triggered (if formerly enabled using the "ENABLEVENTS" command) to indicated the upper layer software that there is a new data packet available to be processed. If one wishes to disable this, specify a character value of -1. Thus, the valid range for that parameter is -1 to 255 (defining disable and all possible byte values). *[Type: signed integer].*
						This defines if the reception path is enabled permanently. Set this to 1 to permanently enable RX path, use 0 for dynamic enabling. Permanently enabled RX path is very important when using the device with the RS-232 option. *[Type: boolean].*

RS485OUT	Device specific command/application only.	This command is used to initiate RS-485/RS-232 data transmission using the serial interface. A specific parameter flag enables or disables local loop-back. For sure that datas to be transmitted have to be specified, too. Global parameters like baud rate etc. are taken from the values formerly set by the "SETRS485CONF" command. So prior to using this command, RS-485/RS-232 interface should be configured properly using the "SETRS485CONF" command.	"OK" if the command was processed successfully or "ERROR" if something went wrong. *[Type: string].*	3	*Command:* RS485OUT=62,113ABCDEF89737546874356536874356534652ABFDCE9432BCDEF897375468743565BCDEF897375468743565368743565346567 2ABFDCE94323AB3,0\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The datas to be transmitted over the RS-485/RS-232 interface of the device in the format of a string containing the datas in hexadecimal manner (ABCD0145, etc.). No hex-preamble or whatever is needed. Thus a byte will occupy 2 characters to be sent to the device. Valid characters are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E and F. Upper and lower case characters are allowed. Maximum data size is 256 byte. The most significant bytes is going to be sent first. *[Type: hexadecimal string].*
						A flag, if local RS-485 feedback is enabled. Means defining a 1 will enable the RS-485 interface to receive its own sent datas, defining a 0 switches that feature off. This is ideal for testing purposes and should normally be switched off in standard operation. *[Type: boolean].*
GETRS485IN	Device specific command/application only.	This command returns the last received bunch of data from the RS-485/RS-232 interface. The data packets are marked with indices to determine if the datas red-back are new datas or prior ones. In general, one does not have to poll that command permanently to watch out for new datas. In addition to receiving datas an asynchronous event notification is dropped to inform the upper layer control software that new data has been arrived (in case asynchronous event notification is enabled). The general interface operation parameters are set using the "SETRS485CONF" command. Thus the "SETRS485CONF" command should be called at the initialization of the device to ensure proper interface operation.	The actual available RS-485/RS-232 datas of the connected device in the format of "RS485_INPUT=%u1,%u2,$DATA$,%b1" where %u1 is going to be replaced by the actual walking index of received RS-485/RS-232 message (starting from 0), %u2 by the RS-485/RS-232 message data offset, $DATA$ by the datas received in hexadecimal style and %b1 by the packet end flag. %u2 and %u3 are important for larger received messages. As the output buffer of the device is not too big, a received huge message and its datas may have to be split into smaller blocks when acknowledged. To red-out the complete message, consecutive execution of this command is needed. Doing so, every execution acknowledge will return different datas with a changed off indicating where the returned datas have to be placed in the total data block. Additionally the final block will have set the packet end flag, indicating that this is the last chunk of data available in buffer and no more consecutive command executions need to be initiated. Vice versa, if the packet end flag is not set, more datas are available to red back. NOTE: After the complete process, datas are cleared from internal buffer and thus get lost! *[Type: unsigned integer, unsigned integer, hex. string, boolean].*	1	*Command:* GETRS485IN=62\r\n *Acknowledge:* RS485_INPUT=1,0,1032547698EFCDBA,0\r\n *Command:* GETRS485IN=63\r\n *Acknowledge:* RS485_INPUT=1,8,1032547698EFCDBA,0\r\n *Command:* GETRS485IN=64\r\n *Acknowledge:* RS485_INPUT=1,16,1032547698,1\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETACCSTATUS	Device specific command/application only.	This command is used to retrieve the actual accelerometer status. This command should be used in reaction to an indicated G-shock event or periodically if asynchronous event notification is not enabled.	The current accelerometer status of the connected device in the format of "ACCSTATUS=$STATUS$" where $STATUS$ is going to be replaced by a bit mask, representing the source of the last G-shock event. Here bit 0 is set when an event on the device X axis has been detected, bit 1 is set when a G-shock on Y axis has been detected and bit 2 is set on an event on device Z axis. *[Type: unsigned int].*	1	*Command:* GETACCSTATUS=65\r\n *Acknowledge:* ACCSTATUS=5\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].

GETACCSENSIFORCE	Device specific command/application only.	This command is used to retrieve the actual accelerometer force sensitivity. This command is helpful for example after device boot sequence to retrieve the accelerometer working force sensitivity.	The current accelerometer g-force sensitivity of the connected device in the format of "ACCSENSIFORCE=$SENSITIVITY$" where $SENSITIVITY$ is going to represent the sensitivity in 1/16g steps meaning a value of 64 represents 4g detection threshold for example. The vaild value range starts at 0 (0g) and reaches up to 127 (8g). *[Type: unsigned int].*	1	*Command:* GETACCSENSIFORCE=66\r\n *Acknowledge:* ACCSENSIFORCE=64\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
GETACCSENSITIME	Device specific command/application only.	This command is used to retrieve the actual accelerometer pulse time sensitivity. This command is helpful for example after device boot sequence to retrieve the accelerometer working pulse time sensitivity.	The current accelerometer pulse time sensitivity of the connected device in the format of "ACCSENSITIME=$SENSITIVITY$" where $SENSITIVITY$ is going to represent the minimum pulse time in 0.5ms steps meaning a value of 16 represents 8ms detection threshold for example. The vaild value range starts at 0 (0ms) and reaches up to 255 (127.5ms). *[Type: unsigned int].*	1	*Command:* GETACCSENSITIME=67\r\n *Acknowledge:* ACCSENSITIME=16\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
SETACCSENSIFORCE	Device specific command/application only.	This command is used to set the working accelerometer force sensitivity. This command should be used previously to watching the accelerometer status to make the device working in the proper and wished sensitivity range. The sensitivity is given in 1/16g steps meaning a value of 64 represents 4g detection threshold for example. The vaild value range starts at 0 (0g) and reaches up to 127 (8g).	"OK" if the command was processed successfully or "ERROR" if something went wrong. [Type: string].	1	*Command:* SETACCSENSIFORCE=68,64\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The new g-force threshold value to set. Valid values reach from 0 for 0g to 127 for 8g. The sensitivity or threshold is given in steps of 1/16g. *[Type: unsigned integer].*
SETACCSENSITIME	Device specific command/application only.	This command is used to set the working accelerometer minimum pulse time sensitivity. This command should be used previously to watching the accelerometer status to make the device working in the proper and wished sensitivity range. The sensitivity is given in 0.5ms steps meaning a value of 16 represents 8ms detection threshold for example. The vaild value range starts at 0 (0ms) and reaches up to 255 (127.5ms).	"OK" if the command was processed successfully or "ERROR" if something went wrong. [Type: string].	1	*Command:* SETACCSENSITIME=69,16\r\n *Acknowledge:* OK\r\n	The current walking command index (has to be incremented every single command to be processed from device). [*Type: unsigned integer, except zero*].
						The new minimum pulse time threshold value to set. Valid values reach from 0 for 0ms to 255 for 127.5ms. The sensitivity or threshold is given in steps of 0.5ms. *[Type: unsigned integer].*

A. Available GIRA Esprit© design frames

Image 34: GIRA Esprit© Glas Schwarz and GIRA Esprit© C Glas Schwarz (with silver bezel).

Image 35: GIRA Esprit© Chrome (with black and silver bezel).

Image 36: GIRA Esprit© Glas Mint and GIRA Esprit© C Glas Mint (with silver and black bezels).

Image 37: GIRA Esprit© Glas Weiss and GIRA Esprit© C Glas Weiss (with silver bezel).

Image 38: GIRA Esprit© Umbra and GIRA Esprit© C Umbra (with silver bezel).

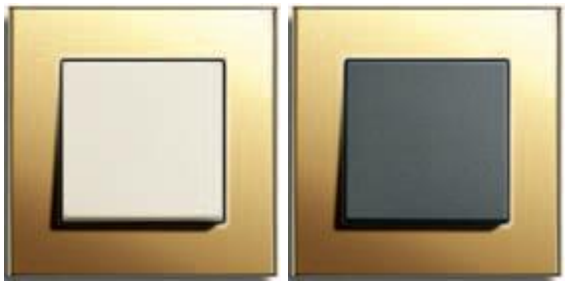

Image 39: GIRA Esprit© Messing (with silver and black bezels).

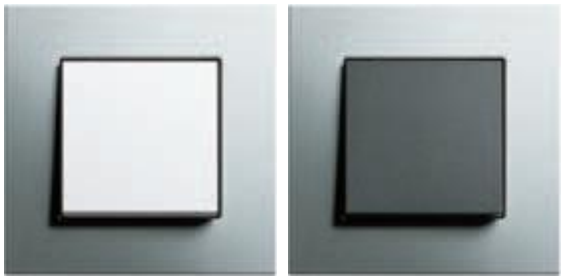

Image 40: GIRA Esprit© Aluminium (with silver and black bezels).

Image 41: GIRA Esprit© Wengeholz (with silver and black bezels).

B. Modified GIRA Esprit© design frame

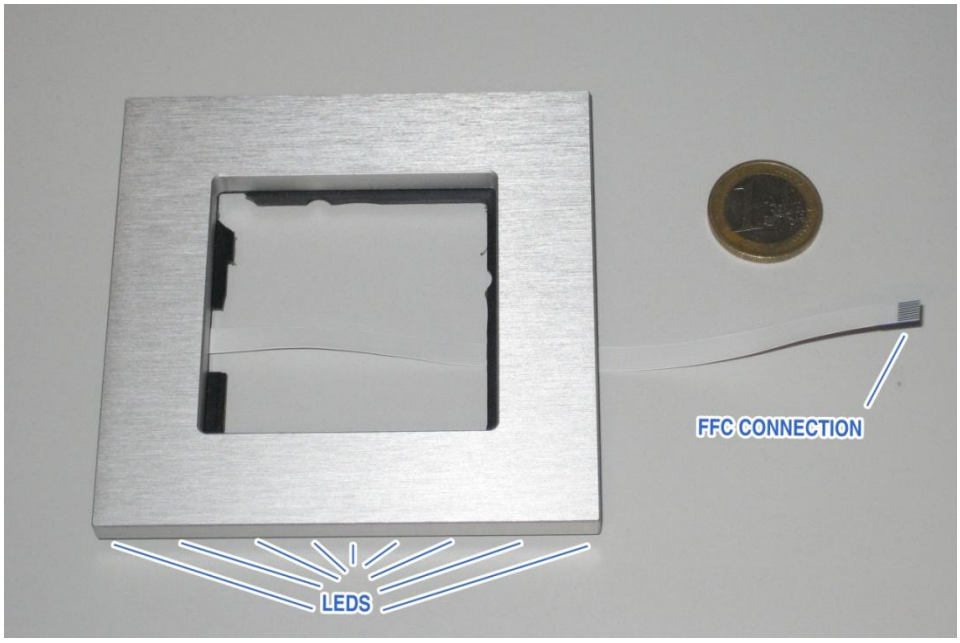

Image 42: Modified GIRA Esprit© design frame front side (aluminum here).

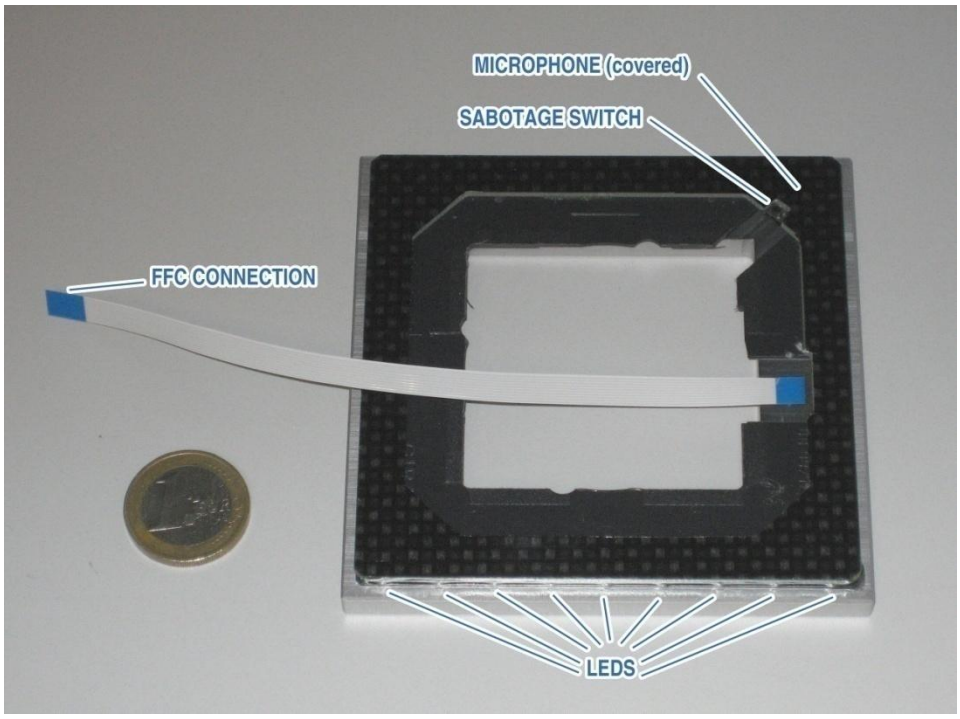

Image 43: Modified GIRA Esprit© design frame backside (aluminum here).

C. Changes to former documentation

C.1 Changes to document "**PalmSecure-iO© Ethernet & USB Version 4.3; System Integrator/OEM-Partner Documentation Ver. 1.0**"

- Color of LED D2 on logic board changed from orange to red/orange.
- For the "SETUSBPOWER" command the initial state of the USB power ports is always OFF, not longer dependent from the environment temperature.

C.2 Changes to document "**PalmSecure-iO© Ethernet & USB Version 4.4; System Integrator/OEM-Partner Documentation Ver. 1.0**"

- Color of LED D4 and D6 on logic board changed from yellow/green to white.
- New commands "ENABLELIFEDETECTION" and "DISABLELIFEDETECTION" added for enabling and disabling life detection (heartbeat detection).
- New commands "GETACCSTATUS", "GETACCSENSIFORCE", "GETACCSENSITIME", "SETACCSENSIFORCE" and "SETACCSENSITIME" for control of G-shock sensor included.
- Command "SETRS485CONF" changed to support RS-232 population option.
- Image 15 changed.

C.3 Changes to document "**PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.0**"

- Because of mechanical modifications on outdoor version, image 17 has been changed.
- Because of mechanical modifications on outdoor version, image 20 has been changed.
- For better readability, image 19 has been changed.

C.4 Changes to document "**PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.1**"

- Because of mechanical modifications on outdoor version, image 17 has been changed.
- Because of mechanical modifications on outdoor version, image 19 has been changed.

C.5 Changes to document "**PalmSecure-iO© Ethernet & USB Version 4.5; System Integrator/OEM-Partner Documentation Ver. 1.3**"

- Chapter 6.11 added for outdoor connector explanation.
- Chapter 6.12 added for outdoor jumper explanation.